



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

The University of Alberta

MANAGING DESIGN INTERACTIONS
WITH CONSTRAINT PROPAGATION
IN AN OBJECT-ORIENTED IC DESIGN ENVIRONMENT

by

Tai A. Ly

A thesis

submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Electrical Engineering

Edmonton, Alberta

Spring, 1989



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-52842-7

Canada

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Tai A. Ly

TITLE OF THESIS:

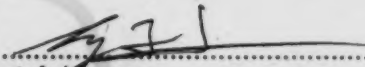
Managing Design Interactions With Constraint Propagation
in an Object-Oriented IC Design Environment

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1989

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

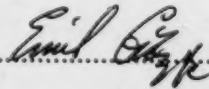
(Signed) 
Permanent Address:
2023 - 52 Street
Edmonton, Alberta
Canada T6L 2G9

Dated 6 April 1989

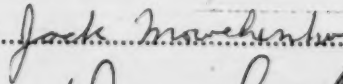
THE UNIVERSITY OF ALBERTA

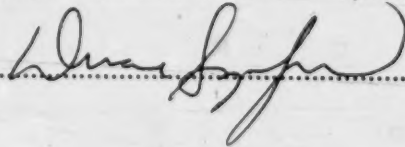
FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Managing Design Interactions with Constraint Propagation In an Object-Oriented IC Design Environment** submitted by **Tal A. Ly** in partial fulfillment of the requirements for the degree of **Master of Science**.



Supervisor





Date

April 6, 1989

ABSTRACT

In IC designs, hierarchical design techniques are widely used to manage the complexity of the design tasks. However, purely top-down or bottom-up design approaches often lead to poor designs because low-level characteristics and high-level specifications do not interact. Better designs are achieved if the bottom-up characteristics of critical components of a design are used to guide the top-down specifications of the other components in the design. This has been referred to as the "least-commitment" strategy of design. To support least-commitment designs, an IC design environment must manage complex interactions among different components in a design.

This thesis investigates constraint propagation as a mechanism for managing such design interactions. Specifically, an object-oriented framework of constraint propagation which manages arbitrary design interactions and supports a least-commitment IC design is presented.

There are two major contributions in this constraint propagation framework. First, it is hierarchical. Constraint networks are organized in hierarchies that parallel the design hierarchies to take advantage of inherent structures in IC designs. As soon as design characteristics become available, they are propagated up the design hierarchy and compared with specifications at various levels. This provides design checking in a global context. Second, this constraint propagation framework is object-oriented. Propagation knowledge is encoded in procedures of abstract data types and inherited by other types in the system. New kinds of constraints are easily added to this framework and arbitrary propagation behavior can be defined by redefining the default procedures.

This framework has been implemented in an integrated design environment named STEM (SmallTalk Environment for Module design) which was developed under a related project. Three sample applications making use of constraint propagation have also been implemented. These are presented in this thesis to demonstrate that constraint propagation is a powerful mechanism for managing arbitrary design interactions in an object-oriented IC design environment.

Acknowledgements

I would like to thank my supervisor, Emil Girczyc, for his guidance and support, and for his heroic work on STEM that provided the platform for this research. I would like to thank Ron Miller, who implemented the Spice interface in STEM and cleaned up a lot of code in the system, and Kent Richardson, who helped implement the initial version of STEM. Last but not least, I thank my wife Christina for her love and support through my graduate school years.

Table of Contents

Chapter	Page
Chapter 1. Introduction	1
1.1. The Least-Commitment Strategy of Design	2
1.2. Integrated IC Design Environments	3
1.3. Constraint Propagation	4
1.4. Overview of Thesis	5
Chapter 2. Related Work	6
2.1. Constraints in IC Designs	6
2.1.1. Electric	6
2.1.2. VEXED	7
2.2. Constraint Propagation in AI	9
2.2.1. EL	9
2.2.2. MOLGEN	10
2.2.3. ThingLab	11
2.2.4. CONSTRAINTS	12
Chapter 3. STEM: an Object-Oriented IC Design Environment	13
3.1. Introduction	13
3.2. Related Work	14
3.3. STEM	15
3.3.1. Model-View-Controller	15
3.3.2. Design Representation in STEM	16
Chapter 4. Constraint Propagation	19

4.1. Constraint Networks	19
4.1.1. Variable Objects	19
4.1.2. Constraint Objects	22
4.2. Constraint Propagation	24
4.2.1. Propagation Scheduling	26
4.2.2. Termination Criteria	29
4.2.3. Constraint Violation	31
4.2.4. Justification for Variable Values	32
4.2.5. Addition and Deletion of Constraints	35
Chapter 5. Integration of Constraints in STEM	38
5.1. Hierarchical Constraint Propagation	39
5.1.1. Implicit Constraint Variables	42
5.1.2. Scheduling of Implicit Constraint Variables	44
5.2. Violation Handling	45
5.3. Constraint Creation	46
5.4. Constraint Editor	47
Chapter 6. Consistency Maintenance	49
6.1. Introduction	49
6.2. Application Interface: Related Work	50
6.3. STEM's Approach to Application Interface	51
6.4. Application Interface	53
6.4.1. Internal Application Programs	54
6.4.2. External Application Programs	56

6.5. Change Management	59
6.5.1. Internal Consistency Maintenance	59
6.5.2. External Consistency Maintenance	61
6.6. Summary	62
Chapter 7. Incremental Design Checking	63
7.1. Signal Types	64
7.2. Bounding Box	70
7.3. Delay	74
7.4. Summary	78
Chapter 8. Module Validation	80
8.1. Module Selection by Generate and Test	82
8.2. Techniques for Efficiency Improvement	83
8.3. Summary	87
Chapter 9. Conclusion	88
9.1. Summary	88
9.2. Discussions	89
9.2.1. Distinctions from Previous Work	91
9.2.2. Comments on STEM	92
9.2.3. Limitations of STEM's Constraint Propagation	93
9.3. Suggestions for Future Work	94
References	97

List of Figures

Figure	Page
3.1 The Model-View-Controller Concept	15
3.2 STEM's two-level model of the design hierarchy	17
3.3 Dual instance variables in cell classes and cell instances	18
4.1 Variable structure	20
4.2 Code segments for setTo:justification:	21
4.3 Code segment for setTo:constraint:justification:	22
4.4 Propagation and satisfaction methods for EqualityConstraint	23
4.5 Constraint propagation in a simple network	25
4.6 Code segment of constraint checking for visited constraints	26
4.7 Scheduling of functional constraints	27
4.8 Code segment for propagating scheduled constraints	28
4.9 Cyclic constraints and their propagation	30
4.10 Code segments for the default violation handler	32
4.11 Code segments for antecedents	34
4.12 Code segments for consequences	35
4.13 Code segment for adding a constraint	36
4.14 Code segment for removing a constraint	37
5.1 Two level hierarchical constraint networks	39
5.2 Example of a hierarchical constraint network	41
5.3 Code segments for implicit constraints propagation	44
5.4 Screenshot of Constraint Editors	48

6.1 Declaration and methods for PropertyVariable	52
6.2 Building a 5-bit Adder with a GraphCompiler	55
6.3 SpiceNet, SpiceSimulation and SpicePlot	58
6.4 User-specified EqualityConstraint in initialization method	60
7.1 Example of bit width constraint violation	66
7.2 Signal type hierarchies	66
7.3 Type hierarchy test methods	67
7.4 Code segments showing signal variable overwrite rule	68
7.5 Signal type constraints due to different cell instances	69
7.6 Screendump showing stretching of io-pins to bounding box	70
7.7 Code segments for bounding box	71
7.8 Subcell bounding box updates parent bounding box	72
7.9 AspectRatioPredicate Definition	73
7.10 The delay model	74
7.11 Dual delay variables in cell class and instances	75
7.12 Hierarchical delay paths and delay constraints	77
8.1 Example of module selection	81
8.2 Test methods	84
8.3 Code for Module Selection	85
8.4 Generic class with properties for search pruning	86
9.1 Two level hierarchical constraint networks	91

Chapter 1. Introduction

In integrated circuit (IC) designs, hierarchical design techniques are widely used to partition a design task into several smaller tasks, each of which is similarly partitioned in turn. In a purely top-down approach, designers start with initial cell specifications, successively decompose the cell into smaller cells and define their interfaces by additional specifications, until the leaf cells of the design hierarchy consist of simple components which can either be designed directly or implemented using library cells. In a purely bottom-up approach, small cells are successively combined into larger cells, whose characteristics are functions of their component cells and interconnects, until a cell which satisfies all design specifications is obtained. These hierarchical design techniques reduce the complexity of design tasks by limiting the interactions among components in the design. This trades off design quality against design time.

With the advent of design environments and efficient design tools, it is faster and easier to iterate the design process and explore different design alternatives. This in turn increases the demand for design environments to do more than traditional design management tasks and provide more methodology support. Most design environments support and encourage hierarchical design methodologies, especially top-down hierarchical design, which offers flexibility in high-level design decisions and is therefore advantageous for design explorations. Unfortunately, these high level decisions are often poorly evaluated because hierarchical designs limit interactions among design components. Specifically, low level design characteristics of components are not considered in higher level specifications of interacting components. This increases the number of design iterations, in which design characteristics are used to correct previous high-level decisions, resulting in new specifications that often require

major redesigns. A better design process would use low-level design characteristics much earlier in the design cycle.

The main motivation behind this research is to provide design environment support for an alternative design methodology, the "least-commitment" strategy of design, in which low-level design characteristics interact with higher level design specifications. In order to do this, the design environment must manage complex interactions among characteristics and specifications of different design components. Towards this end, this thesis investigates a framework of constraint propagation for managing such design interactions in an object-oriented IC design environment.

The following sections provide background information on the least-commitment design strategy, IC design environments and constraint propagation.

1.1. The Least-Commitment Strategy of Design

Mixing top-down specifications with bottom-up design uses low-level characteristics early in the design cycle to guide high-level design decisions. The critical parts of a cell are designed first in considerable detail, which then guide the refinement of design specifications for the rest of the cell. This strategy of deferring design decisions for as long as possible has been referred to as the "least commitment" strategy of design ([Stef81] [Mitc85]).

For example, if the critical delay path of a cell has two subcells, top-down design would assign estimated delay specifications to each of the two subcells such that their sum satisfies the overall delay specification. The two subcells are then designed independently. The fact that the delay of one subcell may be less than its specified delay is not used to relax the delay specification for the

other subcell until design backtracking, when some specifications cannot be met. On the other hand, the "least-commitment" strategy of design would only require that the sum of delays of these subcells satisfy the overall delay specification. The delay specification of each subcell is not "committed" until the characteristic delay of the other subcell is known. The bottom-up characteristics of a subcell are used as soon as they become available to refine (implicit) specifications of other, related subcells.

1.2. Integrated IC Design Environments

Integrated IC design environments aim to provide systems in which design tools can be easily integrated. Most design environments manage a central design database, from which different tools obtain their required data. Due to the different design representations required by different design tools, multiple views of a design must be provided by the environment. For example, while a design rule check (DRC) tool views a design as a set of rectangles in different layers, an electrical simulation tool views a design as a set of interconnected electrical components (e.g., transistors, capacitors, resistors). The interface between the design database and design tools must maintain data consistency among different representations of a design, allow concurrent execution of design tools in the environment (e.g., concurrent editing of a design in two separate windows), and facilitate the development and integration of new design tools.

In order to support a least-commitment design strategy, an integrated IC design environment must manage interactions among specifications and characteristics of different designs. One approach to managing these interactions is to declare explicit constraints among cells, and use propagation of con-

straints to resolve design interactions.

1.3. Constraint Propagation

Constraints are representations of relations that must be true. The power of constraints lies in their capacity to specify consistent value sets for variables. For example, while one equality-constraint can specify that three variables have equal values, the procedural representation of the same relationship is more involved (e.g., checking for equality in the access functions for each variable). Whenever values are assigned to variables, constraints on these variables are checked to ensure that these value assignments maintain a consistent set of variable values. Inconsistent variable values manifest themselves as constraint violations, which require either constraint satisfaction routines, or user intervention, to restore consistency in these variable values.

Constraint propagation is a process in which knowledge about a constraint is used to infer variable values from the values of other variables in the constraint. For example, in order to maintain an equality constraint among a set of variables when a value is assigned to one of these variables, one can simply set all other variables to the same value. These value assignments in turn trigger further value assignments by other constraints, and the effects of the initial value assignment may be propagated to arbitrary distance. This propagation of constraints is inherently an incremental process, in the sense that only those parts of the constraint networks that are affected participate in the propagation. This data directed computation is an important property in an interactive design environment, in which design feedback and system response are both important.

1.4. Overview of Thesis

This thesis presents an object-oriented, hierarchical framework of constraint propagation in an integrated IC design environment. The framework manages arbitrary design constraints, and supports a least-commitment strategy of IC design. It is implemented in STEM (SmallTalk Environment for Module design) [Girc87] using Smalltalk-80 [Gold83]. Sample applications of constraint propagation have been implemented to demonstrate the feasibility of the propagation framework.

The organization of this thesis is as follows. Chapter 2 surveys related work in the fields of integrated design environments and artificial intelligence. Chapter 3 presents a background overview of STEM, with emphasis on STEM's model-view-controller construct and its two level model of the design hierarchy. Chapter 4 describes the details of the constraint propagation framework. Chapter 5 discusses STEM's integration of this propagation framework and the hierarchical implementation of constraint propagation. Chapters 6, 7 and 8 each present a different type of sample application that makes use of constraint propagation in STEM. Finally, chapter 9 summarizes this thesis and concludes with suggestions for future research.

Chapter 2. Related Work

This chapter discusses past research in two major areas related to this thesis. Section 2.1 surveys the use of constraints in IC designs. Section 2.2 surveys constraint propagation systems in artificial intelligence (AI) research.

2.1. Constraints In IC Designs

Many design problems can be formulated as constraint satisfaction problems. Low-level IC designs routinely use linear inequality constraints on positions of layout objects. For example, graph-based compaction algorithms build vertical and horizontal constraint graphs, solve for the maximally constrained paths in the graphs, and then assign node positions to satisfy all constraints in the graphs. Constraint layout languages like Igloo [Pulv87] also incorporate similar linear inequality constraints in procedural layout languages to facilitate the definition of technology independent, parameterizable layouts. Two notable IC design systems that make use of constraints are Electric [Rubi87] and VEXED [Mitic85].

2.1.1. Electric

Electric [Rubi87] is an integrated IC design environment which incorporates many tools for (primarily) layout design. Electric contains a hierarchical, linear inequality constraint system that supports manual design layout. In this system, the user textually or graphically enters linear relations for X, Y coordinates of different geometric objects in the layout. For example, to keep center points of two rectangles at the same "height" in a layout, an equality constraint can be specified on the Y coordinates of these two points. These constraints are treated as properties of wires in the layout, and are stored in the design

database as design data. As the layout is manipulated, a constraint satisfaction routine incrementally solves the constraint networks in a hierarchical manner, performing compaction or decompaction of cells to satisfy connectivity constraints, layout design rules, and layout constraints specified by the designer. A textual report of the constraints on selected objects and the present values that satisfy these constraints is generated on demand. This combines quick, graphical layout entry with flexible, textual parameterization of these layouts using constraints.

However, while layout constraints involving only linear inequalities can be solved with constraint satisfaction, constraints in general include many other types of relations that are not easily solved by any single satisfaction algorithm. For example, the constraint that a component must be centered between two others cannot be expressed in terms of linear inequality constraints in Electric's constraint system [Rubi87].

Electric also provides a Prolog language interface through which users can access a Prolog interpreter. While this provides a gateway for user defined constraints and (Prolog) propagation of these constraints, it does not provide a system wide support for integrating constraint propagation into other design tools in the environment.

2.1.2. VEXED

VEXED [Mitc85] is a knowledge-based consultant for VLSI design. The system makes recommendations for design refinements based on functional specifications. VEXED embodies three categories of knowledge: implementation, control, and causal knowledge. Implementation knowledge encodes the legal implementations for typical design specifications, and control knowledge

guides the design process by ranking the merits of alternative design steps at any given stage. The causal knowledge is used to propagate information about circuit behavior and design constraints from one part of the circuit to another. This causal knowledge is embodied in CRITTER [Kell84], which propagates and checks constraints on value, encoding, and timing of signals.

The CRITTER system is a knowledge-based system for automated critiquing of digital circuit designs. The system works with signal specifications and behaviors that are represented by functional mappings on data-streams. CRITTER analyzes only functional correctness, timing robustness, and speed of a circuit. The input/output mappings of circuit modules are represented as explicit equations for each "feature", and propagation of signal behaviors is accomplished by substitution operations with algebraic simplification. Signal behaviors are propagated forward, while specifications can be propagated backward, by the equation substitution process. The signal behaviors are compared with their specifications and a critique of the circuit is generated.

Besides incorporating CRITTER for managing signal constraints, VEXED also incorporates the BULLDOG [Roac84] system to manage constraints on design layouts. VEXED supports a least-commitment strategy of design by propagating design specifications and behaviors. However, it does not appear to have a unified framework of constraint propagation by which arbitrary extensions to the constraint system may be easily implemented. As a result, each separate subsystem in VEXED would have to replicate general knowledge about constraint propagation, and interactions among different subsystems would require additional knowledge about different constraint systems in these subsystems.

2.2. Constraint Propagation in AI

Propagation of constraints as a general analysis method has been used in both analysis programs (EL [Suss 75, Stall77]) and synthesis programs (SYN [Klee78], MOLGEN [Stef81]). ThingLab [Born77] uses constraint propagation in interactive, graphical simulations of systems. Constraint-based programming languages like CONSTRAINTS [Suss80] facilitate the explicit modeling of interactions among components in "nearly-decomposable" systems.

2.2.1. EL

EL [Stall77] is a knowledge-based system for analysis of electronic circuits. It solves the voltage and current values of electrical nodes in a circuit by the method of constraint propagation. Initially, the system picks a node and introduces variables to be the node's voltage and current. Forward chaining of rules determines the voltages and currents of some other nodes in the circuit in terms of these variables. Whenever there is no more applicable rules, the system picks another "unknown" node and introduces new variables. A "coincidence" occurs when a node which already has a voltage (current) is assigned a new voltage (current). The equation involved in a coincidence can be a tautology, a contradiction, or neither. If it is a tautology, nothing is done; if it is a contradiction, backtracking is invoked to change some assumed states of non-linear devices; if it is neither a tautology nor a contradiction, the system solves for one variable, possibly in terms of other variables, and eliminates this variable throughout the circuit.

Constraint propagation in EL involves the derivation of algebraic expressions for electrical voltages and currents across circuit elements. By propagating expressions with variables, solving local systems of equations, and then

eliminating solved variables throughout the circuit, the system imitates the way human experts analyze electrical circuits without being bogged down by an intractable system of algebraic equations. Furthermore, EL stores dependencies of propagated values for the purpose of explanation as well as dependency-directed backtracking.

2.2.2. MOLGEN

MOLGEN [Stef81] is a hierarchical planner that plans experiments in molecular genetics. It uses constraint posting to manage the interactions among its nearly independent subproblems during the planning process. Constraint posting involves three major operations on constraints: constraint formulation, constraint propagation and constraint satisfaction. MOLGEN dynamically formulates constraints as commitments of the design process. These constraints are propagated by the creation of new constraints. Constraint satisfaction in MOLGEN is achieved either by searching its database for an object that satisfies the constraints, or by solving a subproblem of building such an object.

MOLGEN's constraint propagation operators rely on syntactic matching of the constraints, which leads to its inability to handle constraints other than those in its limited vocabulary. Again, general propagation knowledge is buried in knowledge about specific constraints, making it difficult to incorporate new types of constraints in the system.

2.2.3. ThingLab

ThingLab [Born77] is an interactive system, written in Smalltalk-72 [Gold76], that provides an environment for constructing simulation systems. Each ThingLab object has a list of properties, which describe both the object's internal state and its protocols for sending and receiving messages. Constraint properties of a ThingLab object restrict the behavior of the object. The constraints of an object, like all other properties, are inherited by descendents of the object. When a constraint is added or edited, ThingLab finds those constraints that may be affected, and sends messages to them requesting methods for satisfying them. The system chooses and sets up these methods, and orders the constraints and the objects to which they apply in such a way that each object can be updated in turn to satisfy its constraints in one pass (the "one-pass method") [Born77]. For more complex constraints not directly solvable with this method, the method of assumed states and the relaxation method are tried.

ThingLab provides facilities for incrementally compiling constraints, and a graphic interface controlled by the constraint network. However, since ThingLab does not distinguish between classes and instances, it apparently does not permit the abstraction of constraint networks. Although ThingLab objects inherit properties in an arbitrary abstraction hierarchy, the constraint networks in ThingLab are flat.

2.2.4. CONSTRAINTS

CONSTRAINTS [Suss80] is a language for expressing almost-hierarchical descriptions. It is an interactive system organized around hierarchical networks of constraints. Constraint propagation involves the propagation of values in the constraint networks. The system provides a set of primitive constraints from which compound constraints can be constructed. Each compound constraint then serve as a syntactic abstraction of its corresponding network of constraints. Dependency information is kept with propagated values so that dependency analysis can be used to track down all antecedents of a propagated value as well as all consequences of a value. When simple constraint propagation fails, algebraic manipulation and the method of multiple redundant viewpoints (the slices) [Suss80] are used to resolve the networks of constraints.

However, CONSTRAINTS does not allow definition of new primitive constraints, thus seriously limiting the kinds of constraint networks that can be constructed in the system. Furthermore, while compound constraints provide a concise notation for complex constraint networks, they do not really reduce the computational complexity of the underlying networks. Just before propagation, each compound constraint is dynamically expanded to its component networks of primitive constraints.

Chapter 3. STEM: an Object-Oriented IC Design Environment

This chapter¹ gives an overview of STEM (SmallTalk Environment for Module design) [Girc87], which serves as the platform for this research. STEM is an on-going project at the Electrical Engineering Department in the University of Alberta directed by Dr. Emil F. Girczyc. Its goal is to integrate design tools with manual IC designs using the object-oriented programming paradigm. Interested readers should refer to [Girc87] for additional details on STEM.

3.1. Introduction

IC design environments [Brow83, Girc87] integrate design tools in a uniform and flexible framework for IC design as well as the development of new design tools. Object-oriented programming [Gold83] paradigms are well suited to the development of such design environments because of their abstract data typing and hierarchical inheritance facilities. Abstract data types allow grouping related data and operations on these data into logical entities (classes). This offers a high degree of program modularity, and is well suited to rapid prototyping of software systems. Abstract data types are often organized in a hierarchy of inheritance, by which data and routines may be shared among similar data types. This encourages incremental refinement of abstract data types and facilitates ~~code reuse~~.

¹ This chapter is taken out of the paper on STEM [Girc87]

3.2. Related Work

Typically, an object-oriented IC design environment represents the library version of a cell as a "class" object, which encapsulates all information about the cell such as its characteristics, simulation files, and layout. Individual placements of the cell are represented by instances of this class, and contain fields to represent only that information which differs for each placement of the cell (e.g., location, orientation, and connectivity).

One of the early work in integrated environments, Palladio [Brow83] is an integrated environment that supports different programming paradigms for the development of IC CAD tools. Palladio allows the definition and refinement of "perspectives", which are explicit design representations for different sets of tools. Perspectives are independent design data in the sense that designs in Palladio are defined by a (consistent) set of perspectives, and that each perspective can be part of more than one design.

Fred [Wolf86] is an object-oriented, procedural database designed for the support of VLSI designs. Fred accommodates incomplete modules by storing default values, approximate functions, and dynamic calculations of module properties. Relations are provided as a means to group modules that satisfy certain properties. For example, generic relations specify different implementations of a generic module. Fred also features user defined "dependency", which executes a method whenever a particular message is sent to some modules.

3.3. STEM

STEM (SmallTalk Environment for Module design) [Girc87] is written in Smalltalk-80 [Gold83], and differs from other object-oriented environments primarily in its emphasis on integrating calculated views with user entered views.

3.3.1. Model-View-Controller

STEM is based on the Smalltalk Model-View-Controller (MVC) construct (see Fig. 3.1). A cell in STEM is represented by a single Smalltalk class, which encapsulates all the essential information about the cell. This serves as the model. In order to conserve database storage, it is desirable to have as compact a representation of the model as possible. Views and controllers are unidirectional paths implementing the interface between the user and the model.

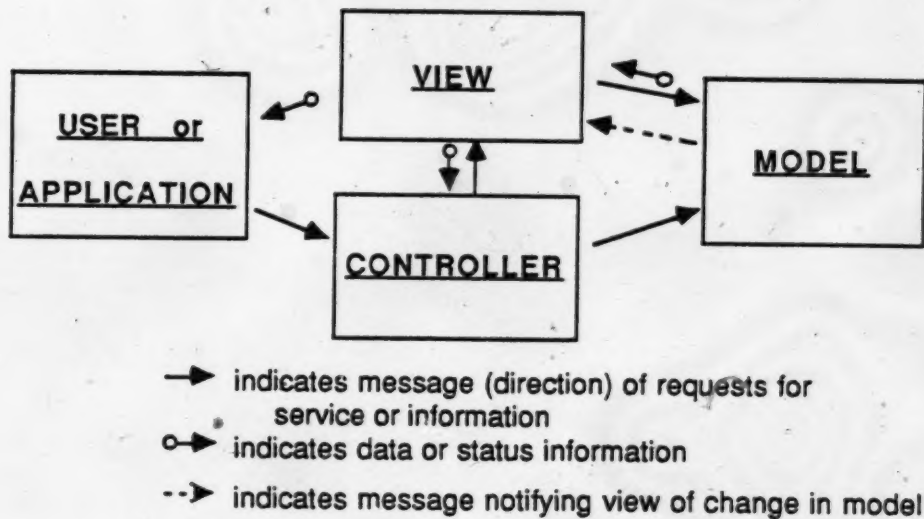


Figure 3.1 The Model-View-Controller Concept

Controllers manage user input. A controller determines the appropriate response to keyboard or mouse input given the current states of the controller,

the model and the connected view. Due to the object-oriented nature of Smalltalk, context dependent response to user input is easily achieved. For example, when the user selects a menu item, the system response is context dependent in two ways: the associations between messages and menu items in the controller and between methods and messages in the model.

Views are representations of the model calculated from data contained in the model. Views translate a portion of the data contained in a model to a format suitable for further processing or display to the user. To accommodate different representations, different views can be employed to format the information contained in a model in different ways. This separation of data (model) and representation (view) allows new views to be added independent of existing views. The model needs to be changed to accommodate a new view only if new data is required to calculate the view.

Views and controllers are linked to the model and to each other through instance variables. This allows views and controllers to obtain data from their model, and context information from each other.

3.3.2. Design Representation in STEM

Viewed from a higher level in the design hierarchy, a cell in STEM is a black box which performs a particular function with a certain interface. The cell interface is defined by a dual declaration of instance variables for each signal, parameter and property defined for the cell. The first variable is declared as an instance variable of instances of the cell class. The value of these variables represent the net connected to a signal or the value of a parameter or property for a particular cell instance used as a component in a larger design. The second instance variable is an instance variable of the class object. This is

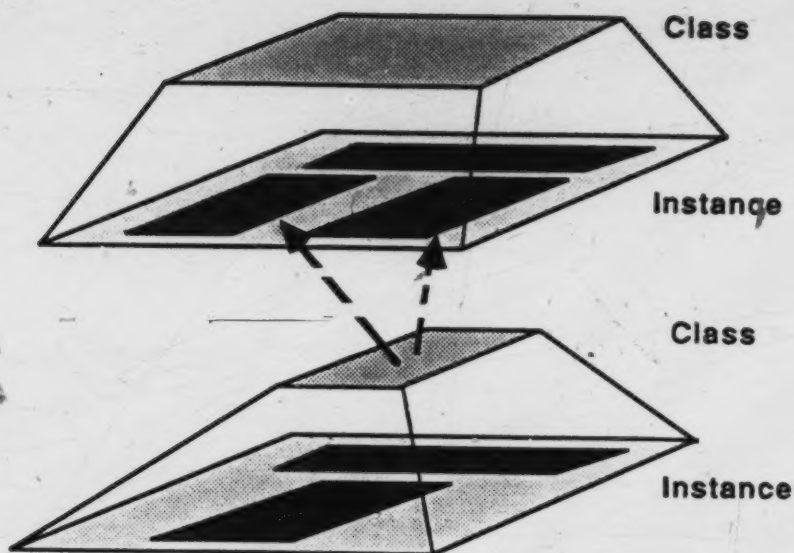


Figure 3.2 STEM's two-level model of the design hierarchy

used to store generic information defining the characteristic limitations of the variable in this cell. For each parameter, this instance variable contains constraints on the range of the parameter, and possibly the default value for that parameter. For each signal, this instance variable contains the data type, electrical type, bit width, and internal connectivities of the signal. This dual declaration of instance variables allows checking to insure that a cell is correctly used within larger designs, and plays an important role in STEM's implementation of hierarchical constraint propagation (Chapter 5).

The Smalltalk inheritance hierarchy is used in STEM to define specialized versions of a cell as subclasses of the cell (e.g., a class ADDER may have many subclasses, each of which being an adder design with a different carry propagation scheme). A subclass inherits all instance variables and methods of

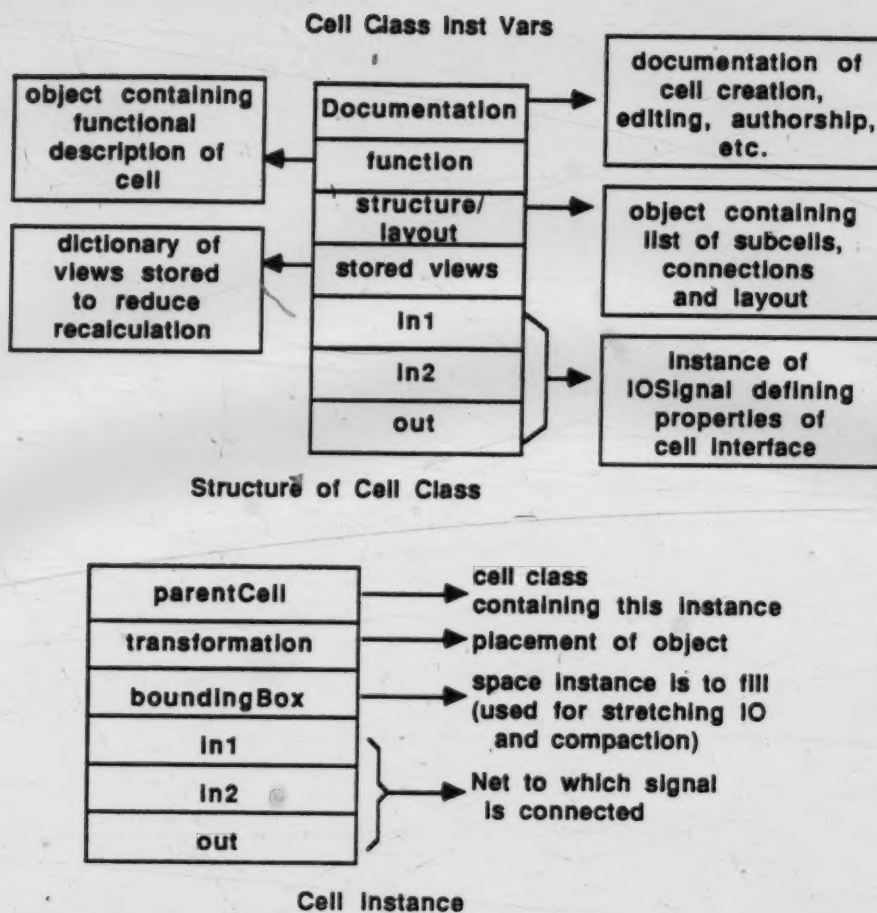


Figure 3.3 Dual instance variables in cell class and cell instances

its superclass (STEM does not use multiple inheritance), and may declare additional instance variables and/or overwrite inherited methods with specialized routines. By using instance variables of classes instead of class variables, values of the inherited variables can be different among different subclasses and their parent class.

Chapter 4. Constraint Propagation

The constraint propagation system incorporated in STEM is organized around constraint and variable objects. The system is designed to provide background coordination for high-level design interactions such as changes in delay, area and signal types among related cells. At present no facility is provided for constraint satisfaction if simple propagation fails. Constraints can be used by both designers and design tools to evaluate alternatives and make design decisions. This chapter describes a general framework for non-hierarchical constraint propagation. While this is developed with STEM in mind, it is general enough to be implemented in any object-oriented integrated design environment. The next chapter describes STEM specific extensions to this general framework that implement hierarchical constraint propagation.

4.1. Constraint Networks

A constraint network is a directed graph in which the nodes consist of "variable" objects and the edges consist of "constraint" objects. Throughout this thesis, "variables" will be depicted with oval shapes and "constraints" will be depicted as rectangles with possibly arrowed lines connected to other "variable" objects.

4.1.1. Variable Objects

A "variable" object (from now on simply "variables") implements an active storage, or "handle", for data so that constraints may be specified on variables independent of their values. Each variable object has a parent-field, a name field, a value field, a constraints field, and a lastSetBy field (Fig. 4.1). The parent field of a variable points to the object that contains the variable, and the

name of the variable identifies the field (instance variable) of its parent that points to the variable. For example, the variable representing the boundingBox (an instance variable) of the class ADDER has ADDER as its parent and "boundingBox" as its name. The parent field and the name field of a variable provide a unique path for identification of the variable object.

VARIABLE Object

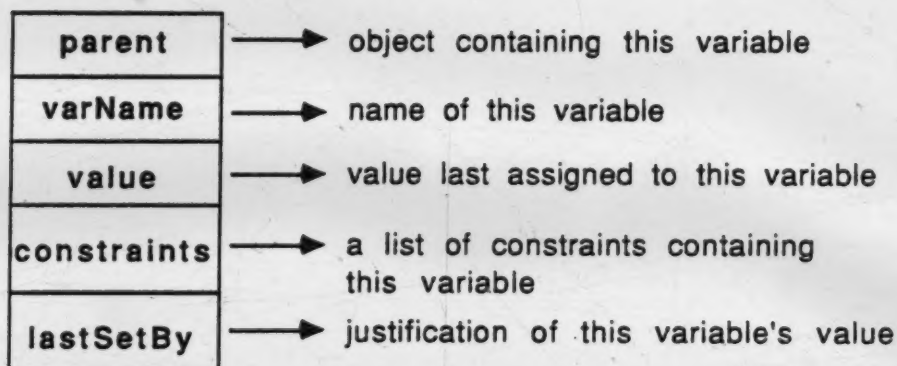


Figure 4.1 Variable structure

The value field of a variable provides storage for the value (or a pointer to the value) of the variable; the constraints field of a variable lists all constraints that reference the variable object; and the lastSetBy field of a variable records the source and possibly dependencies of the current value of the variable (see Section 4.2.4). Variable objects are instantiated during the initialization of their parent objects, and are only removed when their parent objects are removed from the system.

Two basic messages assign values to a variable object: "setTo:justification:"² (Fig. 4.2) and "setTo:constraint:justification:" (Fig. 4.3).

² this is a message denotation of Smalltalk's. The colons in the message indicates the insertion of arguments. "setTo:justification:" accepts two arguments, one following

The first message is used by every object in the system that is not a constraint. The internal method corresponding to this message checks whether constraint propagation is enabled in the system. If it isn't, the method simply sets the variable value to the new value. On the other hand, if constraint propagation is enabled, the method initializes data structures, sets the variable value to the new value, and triggers constraint propagation. A variable propagates by sending the message "propagateVariable:" (using itself as the argument) to all constraints in its constraints field.

Variable methodFor:'assignment'

setTo:value justification:justification

...

self lastSetBy:justification.

self prevValue:value.

status ← self propagate.

...

Variable methodFor:'propagation scheduling'

propagate

self allConstraints do[:constraint

status ← constraint propagateVariable:self.

...]

...

Figure 4.2 Code segments for setTo:justification:

The second message, "setTo:constraint:justification:" is used by constraint objects to assign propagated values to variables during constraint propagation. The internal method corresponding to this message first checks whether any of the termination criteria (see Section 4.2.2) is met. If it is, control is returned; otherwise the method sets the variable value to the new (propagated) value,

"setTo:", and another following "justification:".

and triggers further propagation by sending the message "propagateVariable:" to all its constraints except that constraint which propagated the value to the variable in the first place (Fig 4.3).

Variable methodFor:'assignment'

```
setTo:value constraint:aConstraint justification:justification
... "check for termination criteria" ...
self lastSetBy:(Association key:aConstraint value:justification).
self prevValue:value.
```

```
self allConstraints do:[ :constraint |
    constraint == aConstraint
    ifFalse:[ status ← constraint propagateVariable:self.
              ... ]]
↑ self
```

Figure 4.3 Code segment for setTo:constraint:justification:

4.1.2. Constraint Objects

The "constraint" objects (from now on simply "constraints") specify assertions on variable objects. Each constraint has an "arguments" field, which lists at least one argument which is a variable object. The semantics of a constraint are collectively defined by two methods: "immediateInferenceByChanging" and "isSatisfied". The first method takes the changed variable that activated the constraint as the argument. Generally, this method examines the changed variable and assigns inferred values to other variable arguments in the constraint. The second method tests if a constraint is satisfied by the values of its arguments. For example, for an equality-constraint (Fig. 4.4), the "isSatisfied" method tests that all non-NIL argument values are equal, and the "immediateInferenceByChanging:" method sets all arguments in the constraint (except the source variable) to the value of the source variable.

When the message "propagateVariable:" is sent to a constraint, the constraint is activated. An activated constraint propagates by sending itself the message "immediateInferenceByChanging:" with the changed variable as the argument. Subclasses of constraints customize propagation behavior primarily by redefining the methods "immediateInferenceByChanging:" and "isSatisfied".

Constraint methodFor:'propagate'

propagateVariable:aVariable

↑ self immediateInferenceByChanging:aVariable

EqualityConstraint methodFor:'test'

immediateInferenceByChanging:aVariable

| newValue |

newValue ← aVariable value.

self arguments do:[:arg |

arg == aVariable

ifFalse:[status ← arg setTo:newValue

constraint:self

justification:aVariable.

status isNil ifTrue:[↑ nil]].

isSatisfied

| oldValue |

"check all argument values are equal"

oldValue ← arguments first value.

2 to:arguments size do:[:index |

newValue ← (arguments at:index) value.

newValue = oldValue ifFalse:[↑ false]].

↑ true

Figure 4.4 Propagation and satisfaction methods for EqualityConstraint

4.2. Constraint Propagation

Constraint propagation is actually the propagation of value changes through a constraint network. The value changes occur when a constraint executes a propagation method (procedure) that assigns values to the constraint's argument variables (based on the values of other arguments in the constraint). The propagation comes about when an active variable activates its connected constraints and these active constraints in turn activate their argument variables. Through constraint propagation, the value change in one variable can cause changes in remote parts of the constraint network containing the variable.

For example, Fig. 4.5(a) shows a constraint network containing the variables V1, V2, V3, V4, an equality-constraint and a maximum-constraint. In this diagram, variables are represented by circles and constraints are represented by rectangles. The arrows in the edges indicate directionality of propagation. In this constraint network, the value of V1 must be equal to that of V2, and the value of V4 must equal the maximum of the values of V2 and V3. In the diagram, all variable values satisfy their constraints.

In Fig. 4.5(b), user changes the value of V1 to 9. This triggers constraint propagation (indicated by thick arrow). The variable V1 activates the equality-constraint, which propagates by assigning V2 the value 9. This activates V2, which in turn propagates by activating the maximum-constraint. Finally, the maximum-constraint recalculates the maximum of V3 and V2 and assigns V4 the value 9.

The basic process of constraint propagation implemented in this thesis can be characterized as a depth-first traversal of a constraint network. Depth-first traversal is used instead of breadth-first to reduce the amount of intermediate state information that must be kept during the traversal. The traversal starts

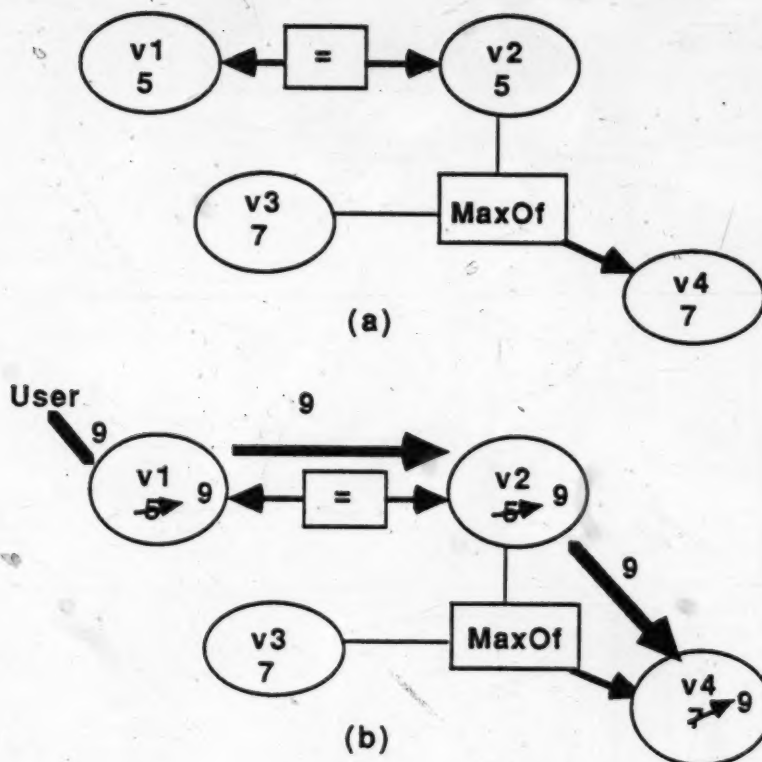


Figure 4.5 Constraint propagation in a simple network

with the (external) assignment of a variable by the message "setTo:justification:", and successively spreads to all constraints of the variable. Each of these constraints in turn performs its inferences ("immediateInference-ByChanging:"), and assigns values to additional variables by the message "setTo:constraint:justification:". Each of these value assignments may trigger further constraint propagation. Throughout the propagation process, control

alternates between variables responding to the message "setTo:constraint:justification", and constraints responding to the message "propagateVariable:". All visited variables and constraints are recorded during propagation. When propagation ends without error, all visited constraints are checked for constraint violations by sending the message "isSatisfied" to each visited constraint (Fig. 4.6).

Variable methodFor:'assignment'

setTo:value justification:justification

```
...
visitedConstraints do:[:constraint |
    constraint isSatisfied ifFalse:[↑ self visitedConstraintViolation]].
...
```

Figure 4.6 Code segment of constraint checking for visited constraints

4.2.1. Propagation Scheduling

When a changed variable sends the message "propagateVariable:" to a constraint, the constraint usually propagates inferred values of other argument variables as soon as possible. This is because the direction of propagation is dependent on the changed variable that initiated this propagation. In an equality-constraint involving A and B, for example, the value of the changed variable is propagated to the other variable. If A changes, B is set to A's new value; if B changes, A is set to B's new value. Propagation is performed in the first-come-first-served manner, in order that the directionality of constraint propagation be correct.

However, there are constraints whose direction of propagation is independent of the variable that changed. These are the functional-constraints, so named because they can often be expressed as a mapping of a tuple of

variables onto another tuple of variables. For example, a unidirectional, arithmetic constraint expresses one variable (the functional variable) as an arithmetic function of other variables or constants. The inference method of the constraint calculates the functional value from values of the argument variables, and assigns the resulting value to the functional variable in the constraint. For these constraints, propagation can be delayed until all argument variables have had a chance to change. This reduces redundant calculations of transient results.

An agenda object schedules the propagation of these constraints. In response to the message "propagateVariable:", a functional constraint schedules itself for propagation by adding itself to an agenda named #functionalConstraints (Fig. 4.7).

FunctionalConstraint methodFor:'propagate'

```
propagateVariable:aVariable
visitedConstraints add:self.
(self permitChangesByVariable:aVariable)
ifTrue:[
    agendaScheduler scheduleConstraint:self
    variable:nil
    onAgendaNamed:#functionalConstraints]
```

FunctionalConstraint methodFor:'test'

```
permitChangesByVariable:aVariable
"returns false iff aVariable is my result variable"
↑ (aVariable == self resultVariable) not
```

Figure 4.7 Scheduling of functional constraints

An agenda is implemented as a first-in-first-out queue that does not have duplicate elements. New entries are added to the end of an agenda provided similar entries are not already in the queue.

In general, there may be more than one agenda in the system, each with a different, fixed priority. Constraints scheduled in agendas are propagated one at a time, according to the priorities of the agenda they are scheduled in. This multi-queue, fixed priority scheduling is implemented in the variable method "setTo:justification:", which, after initial propagation of un-scheduled constraints, iteratively selects a scheduled constraint from the highest-priority agenda that is not empty and propagates it, until all agendas are empty (Fig. 4.8). Each propagation of a scheduled constraint may trigger additional propagation and schedule more constraints in the agenda. With this additional degree of control, propagation can be made more efficient by assigning higher priorities to critical constraint types such as equality-constraints and update-constraints.

Variable methodFor:'propagation scheduling'

propagate

```
self cllConstraints do[:constraint
  status ← constraint propagateVariable:self.
  ... ]
```

```
[scheduledEntry ← agendaScheduler removeHighestPriorityScheduledEntry.
 scheduledEntry isNil]
```

```
whileFalse:[
```

```
  status ← (scheduledEntry key)
  propagateScheduledVariable:(scheduledEntry value).
```

```
... ].
```

AgendaScheduler methodFor:'access'

removeHighestPriorityScheduledEntry

```
| agenda |
```

```
"returns first entry of the highest priority non-empty agenda"
```

```
(self priorityList) do[:agendaName |
```

```
  agenda ← self agendaNamed:agendaName.
```

```
  agenda isEmpty ifFalse:[ ↑ agenda removeFirst]].
```

```
↑ nil
```

Figure 4.8 Code segment for propagating scheduled constraints

4.2.2. Termination Criteria

The wavefront of constraint propagation may stop at one of two cases:

1. The propagation method of a constraint does not assign values to any variable,
2. The current value of a variable agrees with the propagated value of the variable,

Constraint propagation terminates as soon as a constraint violation is detected. A constraint violation is detected during propagation in one of two cases:

1. The variable to be assigned a value has already been visited in the current propagation, but has a different value than the propagated value,
2. The current value is user-specified and cannot be modified by constraint propagation, but has a different value than the propagated value.

If no violation is detected during constraint propagation, then the process of constraint propagation terminates when the non-functional constraint propagation is exhausted and the agenda is empty. This termination is guaranteed in STEM by requiring that no variable be allowed to change its value twice during propagation. With this rule, STEM prohibits cyclic propagation of constraints but not the use of cyclic constraints. For example, in Fig 4.9, three addition constraints form a cyclic constraint network which cannot be satisfied. Suppose V1 is assigned a value of 10, constraint propagation (indicated by bold arrows) assigns a value of 11 to V2 and then a value of 14 to V3, and tries to assign a value of 16 to V1. However, since V1 has already changed its value once during this round of constraint propagation, this triggers a constraint violation.

This one-value-change rule is enforced with a global dictionary, VisitedCon-

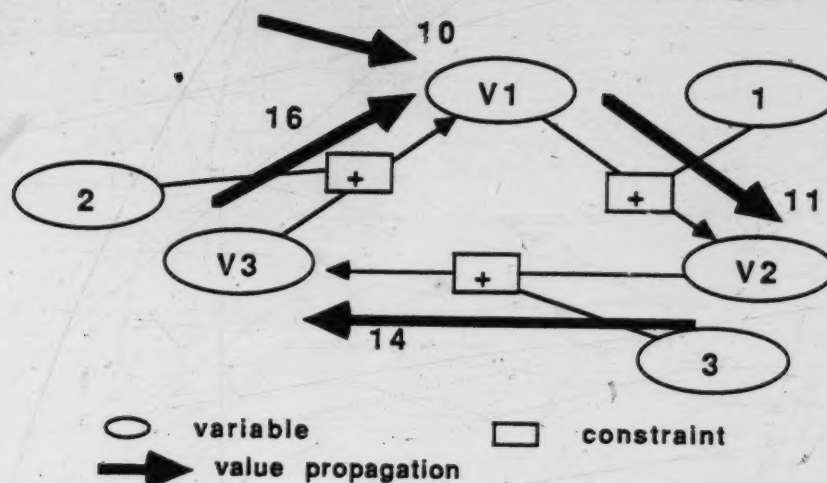


Figure 4.9 Cyclic constraints and their propagation

straintsAndVariables, of all visited constraints and variables during the propagation. This global dictionary is emptied in the initialization phase of the method "setTo:justification:". Subsequently, every time a constraint propagates a value to a variable with a value different than the propagated value, the variable checks in the dictionary to see if the variable has been visited previously. If it has, a constraint violation is detected, and propagation is terminated. On the other hand, if the variable has not been visited previously, the variable adds its current state to the dictionary of visited variables, changes its value, and initiates further constraint propagation. The global dictionary records the previous state of all visited variables so that these variables can be restored to their original states if a constraint violation is detected in the new variable values.

4.2.3. Constraint Violation

Although propagation methods in the constraints generally maintain consistency of the constraints, constraint violations are inevitable because only local information (in the constraint arguments) is considered in constraint propagation, and because invalid user-assigned variable values manifest themselves as constraint inconsistencies. Properly managed, constraint violations provide valuable feedback to the designers, not only on the presence of design violations, but on the causes of these violations as well.

Constraint violation may be detected in the course of propagation, as soon as a propagated value disagrees with a variable value and an overwrite is not possible (either because the propagated value has no overwrite privilege over the existing value, or because the variable has been visited earlier in the propagation). Even if constraint propagation finishes without any violation, constraint violations can still exist. As a final check, the message "isSatisfied" is sent to all visited constraints to detect violations.

When a constraint violation is detected, the violation handler of the constraint is executed. The default (inherited) handler issues warning messages in a default text window, and restores the constraint networks to their original states (i.e., before constraint propagation) by restoring the states of all visited variables (Fig 4.10). However, different types of constraints may replace this with different violation handlers. For example, a constraint editor in STEM may be invoked to help designers examine and debug the constraint networks.

Variable methodFor:'violation handling'

violation

"restore prev values"
self restore.

...

restore

"restores prev values for all variables"
visitedVariables associationDo[:assoc |
var ← assoc key.
var lastSetBy:assoc value key.
var prevValue:assoc value value].

...

Figure 4.10 Code segments for the default violation handler

4.2.4. Justification for Variable Values

When variables are assigned values, the justification of these values are recorded in the "lastSetBy" field of the variables. The justification of a value assignment is formulated by the sender of the assignment message and is passed to the variable as one of the arguments in the assignment message (i.e., the argument following "justification:" in both "setTo:justification:" and "setTo:constraint:justification:"). A justification can either be a symbol indicating a source external to the constraint networks or a key-value pair for a propagated value. The symbols #USER and #APPLICATION are currently used as justifications for user specified and calculated values respectively. The justification of a variable's value determines if constraint propagation can overwrite the variable with a different propagated value. The default overwrite rule in the system is that user specified values have higher priority over propagated and calculated values. However, subclasses of variables can redefine this rule of precedence. For example, variables can recognize different strengths of constraints, and allow one type of constraints to overwrite values

from another type of constraints, but not the other way around. This is not done currently.

For propagated values, the justification is a key-value pair of two elements. The first element is the source constraint which propagated the value to this variable. The second element is a dependency record for this propagated value. This dependency record is formulated by the source constraint during propagation and analyzed by the same constraint during dependency analysis. Each dependency record contains data that enable the source constraint to trace all variable values that are responsible for the propagated value. Since dependency records are only interpreted by the constraints that formulate them, they vary greatly among different types of constraints. For example, an equality-constraint sets up a dependency record containing only the single variable that activated the constraint, but a functional constraint sets up a null dependency record since it is implicitly understood that the functional variable depends on every arguments of the functional constraint.

With source constraints and dependency records, dependency graphs in the constraint networks can be constructed for dependency analysis. Dependency analysis can be used to search for all variables and constraints responsible for a propagated value. This is achieved primarily with two methods: "antecedents:" in variables, and "antecedents:ofVariable:" in constraints (Fig. 4.11). These two routines recursively call each other, performing a backward traversal of the dependency graphs, until all antecedent variables and constraints of a variable are found.

Dependency analysis can also be used to search for all variables that depend on a specific variable value. This is achieved with two similar methods: "consequences:" in variables, and "consequences:ofVariable:" in constraints

Variable methodFor:'dependency analysis'

antecedents:aSet

```
(aSet includes:self) ifTrue:[ ↑ aSet ].
aSet add:self.
self isDependent ifTrue:[
  aConstraint ← self lastSetBy key.
  aConstraint antecedents:aSet ofVariable:self ].
↑ aSet
```

Constraint methodFor:'dependency analysis'

antecedents:aSet ofVariable:aVariable

```
aSet add:self.
"default method will just test all arguments one by one"
dependencyRecord ← aVariable lastSetBy value.
self allConstraintVariables do:[ :arg |
  arg == aVar ifFalse:[
    (self testMembershipOf:arg inDependency:dependencyRecord)
    ifTrue:[ arg antecedents:aSet ]]].
↑ aSet
```

testMembershipOf:aVar inDependency:dependencyRecord

```
"each subclass redefines its own"
self subclassResponsibility
```

Figure 4.11 Code segments for antecedents

(Fig. 4.12). These two routines also recursively call each other, performing a forward traversal of the dependency graphs, until all consequences of a variable's value are found.

While dependency analysis is useful in examining the constraint networks, it is essential in erasing inferences invalidated by deletion of constraints in the networks. When a constraint (variable) is removed from a network, all variable values dependent on the constraint (variable) become unjustified, and should be erased. This erasure is facilitated by dependency analysis, which finds all consequent variables of the constraint (variable). In an interactive design environment, constraint networks are frequently edited. The efficiency with which constraint (variable) removals can be accomplished with dependency analysis

justifies the storage overhead for dependency records in variables.

Variable methodFor:'dependency analysis'

```
consequences:aSet
(aSet includes:self) ifTrue:[ ↑ aSet ].
aSet add:self.
self constraints do:[ :aConstraint |
  aConstraint consequences:aSet ofVariable:self ].
↑ aSet
```

Constraint methodFor:'dependency analysis'

```
consequences:aSet ofVariable:aVariable
self allConstraintVariables do:[ :arg |
  arg == aVariable ifFalse:[
    arg lastSetBy key == self "arg is set by me"
    ifTrue:[ "then test if aVar is in dependencyRecord of arg"
      dependencyRecord ← arg lastSetBy value.
      (self testMembershipOf:aVar inDependency:dependencyRecord)
      ifTrue:[ arg consequences:aSet ]]]]
```

Figure 4.12 Code segments for consequences

4.2.5. Addition and Deletion of Constraints

When constraint networks are edited (i.e., when constraints or variables are added to or removed from the networks), constraint propagation must be performed to adjust variable values to changes in the networks. However, since no variable has changed value by network editing, the normal propagation trigger is not activated. Therefore, a separate triggering mechanism is developed for constraint editing.

When a constraint is added to a variable, the constraint re-initializes its variable arguments (Fig. 4.13). The argument variables are grouped into three lists: "userSpecified", "constraintDependent", and "otherIndependents". The first and second lists contain variables whose justifications are #USER and a

Variable methodFor:'add-remove constraints'

addConstraint:aConstraint

```
...
self constraints add:aConstraint.
aConstraint basicAddArgument:self.
status ← aConstraint reInitializeVariables.
...
```

Constraint methodFor:'add-remove arguments'

reInitializeVariables

```
...
status ← self rePropagate.
...
```

rePropagate

```
... organize argument variables into three lists ...
list addAll: userSpecified.
list addAll: dependents.
list addAll: otherInDependents.
list do:[:arg |
    keyAbsent ← visitedVariables at:arg
                putIfAbsent:(Association key:arg lastSetBy
                                     value: arg prevValue).
    keyAbsent ifTrue:[ status ← arg propagateAlongConstraint:self.
                      status isNil ifTrue:[ ↑ nil ]]]
```

Variable methodFor:'propagation scheduling'

propagateAlongConstraint:aConstraint

```
status ← aConstraint propagateVariable:self.
status isNil ifTrue:[ ↑ self inPropagationViolation ].
[scheduledEntry ← agendaScheduler removeHighestPriorityScheduledEntry.
 scheduledEntry isNil]
whileFalse:[
    status ← (scheduledEntry key)
              propagateScheduledVariable:(scheduledEntry value).
    status isNil ifTrue:[ ↑ self inPropagationViolation ].
    ↑ self
```

Figure 4.13 Code segment for adding a constraint

constraint object, respectively, while the third list contains variables with other types of justifications (e.g., #APPLICATION). Beginning with "userSpecified" and ending with "otherInDependents", these argument variables are successively sent the message "propagateAlongConstraint:" with the edited constraint

as the argument. In response to this message, each variable propagates its value to the edited constraint. This gives a chance to all variable arguments in the edited constraint to assert and propagate their values, in order of their precedence. If constraint propagation finishes without violations, all visited variables are explicitly checked for constraint violation by sending the message "isSatisfied" to them.

When a constraint is removed from a variable, all propagated values that depend on this constraint/variable pair are reset to NIL. This is accomplished by dependency analysis (Fig. 4.14). If the variable's value is changed by removing the constraint, then all consequences of the variable have to be reset. If the variable's value does not change when the constraint is removed, then all consequences of the constraint that depend on this variable have to be reset. In either case, the removed constraint then re-initializes its remaining argument variables.

Variable methodFor:'add-remove constraints'

```
removeConstraint:aConstraint
self constraints remove:aConstraint.
aConstraint basicRemoveArgument:self.
self lastSetBy key == aConstraint
ifTrue:[
    "my value is last set by aConstraint,
    reset me and all my consequences"
    self reset.
    self variableConsequences do:[ :aVar | aVar reset ] ]
ifFalse:[
    "reset all variables that are consequences of me
    propagating through aConstraint"
    set ← Set new.
    aConstraint consequences:set ofVariable:self.
    set do:[ :aVar | aVar reset ].
    "rePropagate aConstraint"
    aConstraint reinitializeVariables
```

Figure 4.14 Code segment for removing a constraint

Chapter 5. Integration of Constraints In STEM

This chapter presents the way in which the constraint propagation framework is integrated with design objects in STEM. In order to take full advantage of inherent structures in these design objects, STEM specific extensions are added which implement hierarchical constraint propagation. With hierarchical constraint propagation, constraint networks in each cell class are separate from those in other cell classes, and yet constraint propagation can spread from one network to another up and down a hierarchy which parallels that of the design objects. Basically, hierarchical constraint propagation occurs when propagation in the constraint network of one cell activates additional propagation in constraint networks of cells containing instances of the original cell or cells whose instances are part of the original cell.

Hierarchical constraint propagation is achieved with STEM's dual instance variables in the class and instance definitions of cells. These dual variables provide the implicit propagation links between otherwise isolated constraint networks at the cell class level and cell instance levels. Hierarchical constraint propagation reduces the amount of redundant computation in composite networks. For example, Fig 5.1 shows three isolated constraint networks that are linked by implicit constraints (indicated by the bold dotted arrows) between dual instance variables in the cell class and cell instances. When constraint propagation in the lower network changes the value of the variable in the cell class, hierarchical propagation changes the variable values in the two cell instances and triggers additional propagation in the two upper level networks. Without hierarchical constraint propagation, the lower level constraints in Fig. 5.1 would be propagated twice: once for each of the two upper level networks containing them.

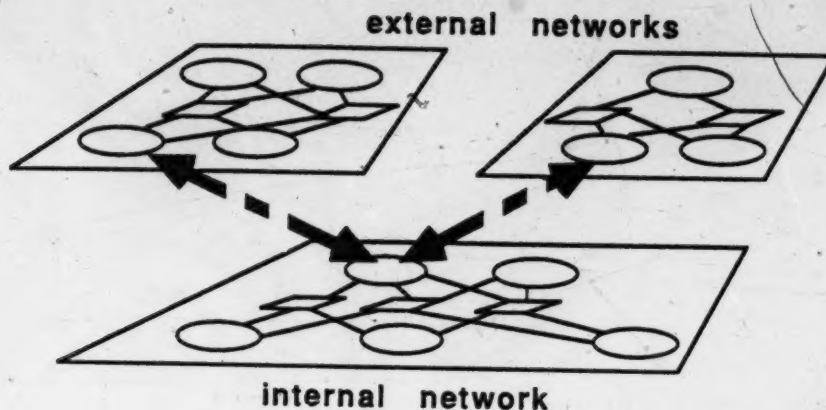


Figure 5.1 Two level hierarchical constraint networks

In order to incorporate constraint propagation in STEM cells, all instance variables of cell classes and instances are initialized to appropriate variable objects on instantiation and all access methods of instance variables are modified accordingly. Constraint creation and violation handling are also modified for STEM, and a simple constraint editor is added to improve the user interface.

5.1. Hierarchical Constraint Propagation

Constraint networks in STEM are organized around the design hierarchy to take advantage of STEM's power of abstraction. The dual declaration of instance variables in STEM forms the basis of hierarchical constraint networks. The values assigned to variables of a cell class represent the characteristics of the internal structure of the cell. The constraints on these variables represent conditions imposed by designers and the cell's internal structure on the corresponding characteristics of the cell. On the other hand, constraints on

variables of the cell instances represent conditions on the cell imposed by the environments in which these cell instances are used.

For example (Fig. 5.2) when a designer first designs an eight-bit ADDER, a delay constraint of "120ns or less" may be specified on the appropriate delay variable of the cell class. As the internal structure of the ADDER is designed, constraint violation is triggered if a delay value greater than 120ns is propagated to this delay variable. The delay constraint at the class level therefore constrains the internal design of this cell. Subsequently, an instance of the ADDER cell may be used in an ACCUMULATOR cell, built by cascading an 8-bit REGISTER to an ADDER, which has a overall delay constraint of "160ns or less". From the interconnects of the ADDER instance, a constraint network involving the delay variable of the ADDER instance is instantiated. If the characteristic delay of the REGISTER instance is 60ns and that of the ADDER instance is 110ns (after adjustment for loading), then a constraint violation is triggered.

There are therefore two levels of constraint networks for a cell: the internal and external constraint networks (Fig 5.1). These correspond to the internal and external views of a cell. Internal constraint networks of a cell involve variables of the cell class, and represent conditions that the cell's internal structure must satisfy. The external constraint networks of a cell generally involve variables of cell instances, and represent (indirectly) conditions which the instances must satisfy to be used in their respective environments. Whenever design characteristics become available in a cell class, they are checked with internal constraint networks of the cell and then propagated to the instances and their corresponding external constraint networks. This updates characteristics of larger designs and triggers further constraint propagation. Hierarchical con-

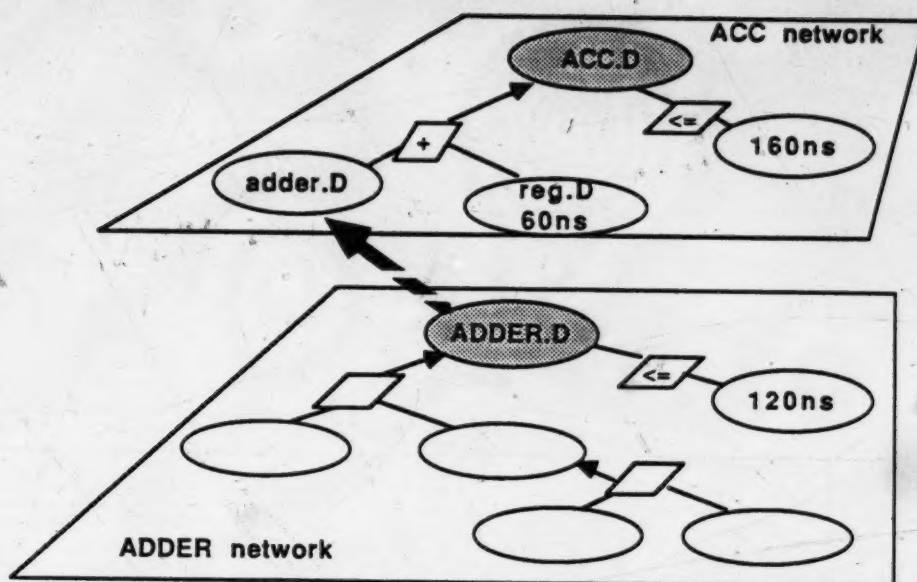


Figure 5.2 Example of a hierarchical constraint network

straint propagation reduces the amount of computation since internal constraint networks are propagated only once even though there may be a large number of cell instances. This takes advantage of the hierarchical structures of IC designs in STEM.

To implement this hierarchical constraint propagation, the concept of an implicit constraint is introduced. An implicit constraint is a procedural implementation of a constraint, or a "hard coded" constraint. For the purpose of hierarchical constraint propagation, implicit constraints are embedded in the dual variables of cell classes and cell instances. These variables respond to propagation messages as if they were constraint objects. This is implemented in a subclass of Variable named "ImplicitConstraintVariable". Two subclasses of ImplicitConstraintVariable are defined for use as instance variables of STEM objects.

Instances of class "InstanceInstVar" are used for instance variables of cell instances, and those of the class "ClassInstVar" are used for instance variables of cell classes. In addition, a new agenda named #implicitConstraints is added to schedule these variable-constraints for hierarchical constraint propagation. The next two sections describe implicit constraint variables and the scheduling of their propagation.

5.1.1. Implicit Constraint Variables

A constraint can be made implicit if the links between the constraint and its argument variables are sufficiently well defined that these links can be proceduralized. Due to the dual declaration of instance variables in STEM cells, an instance of "InstanceInstVar" is an implicit constraint on its corresponding instance of "ClassInstVar"; and an instance of "ClassInstVar" is an implicit constraint on all of its corresponding instances of "InstanceInstVar". These variable-constraints play the roles of both variable and constraint, by being the descendents of "Variable" and responding to propagation messages like "isSatisfied" and "propagateVariable:" that constraint objects respond to.

Whenever a value assignment is made to an instance of "InstanceInstVar" or "ClassInstVar", the variable object sends the message "propagateVariable:" to all of its explicit constraints (i.e., constraint objects in its list of constraints), as well as all of its implicit constraints (as returned by the method "implicitConstraints"). All visited constraints, including the implicit constraints, are then sent the message "isSatisfied" to check for constraint violation. Descendents of "ClassInstVar" and "InstanceInstVar" may or may not propagate to their implicit constraints. Two categories of descendents of these variables can be identified: parameters and properties.

For parameters in a cell, the instance variables in the cell class characterize the range of the parameter values that can be handled by the cell. The corresponding instance variables in the cell instances specify the actual parameter values in each use of the cell. Whenever a parameter variable in a cell instance is assigned a new value, it checks with its corresponding variable in the cell class that the value is valid for that parameter of this cell. On the other hand, whenever a parameter variable in a cell class is assigned a new range, it checks with all of its corresponding parameter values in the cell instances that these values satisfy the new range for this parameter. Except for default values of parameters that may be propagated from class parameter variables to instance parameter variables, no constraint propagation is performed between corresponding parameter variables.

For properties of a cell, the instance variables in the cell class characterize the nominal values of these properties. Their dual variables in the cell instances specify the values for these properties adjusted to the contexts of each cell instance. Whenever a new value is assigned to a property variable of a cell instance, the variable must check with its corresponding variable in the cell class that the new value is consistent with the cell property (e.g., the bounding-box of a cell instance must not be smaller than that of the cell class). On the other hand, whenever a new value is assigned to a property variable of a cell class, the variable must check with all of its corresponding variables in the cell instances that their values are consistent with the new property value. Property values may propagate from classes to instances, perhaps with some adjustments, but never from instances to classes. For example, the delay property values of a class are propagated to corresponding delay variables in the cell instances after adjustments for local loading conditions. However, delay variables in the cell instances do not propagate to their dual delay variables in

the cell class (see Chapter7).

5.1.2. Scheduling of Implicit Constraint Variables

In order to accommodate hierarchical constraint propagation, an agenda named `#implicitConstraints` is used to schedule propagation of implicit constraints (Fig. 5.3). The objective is to delay propagation of implicit constraints up (or down) the design hierarchy as much as possible so as to take advantage of the locality constraints in designs. This is achieved by assigning the lowest priority to the `#implicitConstraints` agenda. The end result is that hierarchical constraint propagation tends to completely propagate constraint networks in one level of the hierarchy before propagating constraint networks in another level of the hierarchy.

```
ImplicitConstraints methodFor:'propagate'

propagateVariable:aVariable

visitedConstraints add:self.

self permitChangesByImplicitPropagation
ifTrue:[
    agendaScheduler
    scheduleConstraint:self
    variable:aVar
    onAgendaNamed:#implicitConstraints]

ImplicitConstraints methodFor:'test'

permitChangesByImplicitPropagation
"defaults to true"
↑ true
```

Figure 5.3 Code segments for implicit constraints propagation

5.2. Violation Handling

When a constraint violation occurs in STEM, it is not sufficient to just restore all variables to their original values, since eliminating the violation involves reversing the design step that has led to the violation in the first place. This design step may be as simple as a value assignment to a variable, or as complicated as the compilation of a parameterized module. In order for design tools to take advantage of the background checking offered by constraint propagation, the validity of a value assignment is returned by the variable assignment methods "setTo:justification:" and "setTo:constraint:justification:". If a value assignment is invalid, NIL is returned by these methods; otherwise, a non-NIL value is returned. This validity feedback enables design tools to detect constraint violations caused by value assignments, so that appropriate steps may be taken to remove these violations. Similarly, the access methods that introduce constraints return NIL if constraint violations occur as a result of the new constraints, hence providing validity feedback to design tools.

When a constraint violation occurs, the violation handler of the constraint is invoked. A common violation handler in STEM prompts the user for one of two alternative actions: "debug" or "proceed". If "debug" is selected, a constraint debugger associated with the violated constraint is invoked. The default constraint debugger is a constraint editor, although different constraints may use different, specialized debuggers. If the designer selects the "proceed" option, a warning message is printed and all changed variables are restored to their original states before continuing constraint propagation.

5.3. Constraint Creation

Constraints in STEM are added and removed either by the access methods in cell objects, or by designers through a constraint editor. For example, before a cell is designed, a designer can specify the io-signals of the cell, and place delay constraints between pairs of io-signals (e.g., delay from A to B must not be longer than 100ns). As the internal cell design progresses, delay constraints are instantiated when subcells are added and removed when subcells are removed by the "addCell" and "removeCell" methods, respectively. Constraint violation occurs if the internal delay network produces a delay time longer than the user-specified delay constraint, and the designer is warned. To continue the design, the designer can either disable constraint propagation, change the internal cell design to achieve a shorter delay, or relax the violated delay constraint on the cell.

Constraint propagation can be disabled in STEM by setting a global disable flag, "CPSwitch". This status flag is checked by the "setTo:justification:" methods of all variables. These methods perform simple value assignments without initiating constraint propagation or constraint checking when the flag is set to #On. Constraints are still added and removed by access methods and constraint editors when the flag is set. However, the local constraint propagations associated with new constraints are disabled when constraint propagation is disabled. This ability to disable constraint propagation is desirable if designers embark on extensive design revisions which, while producing massive design violations in the process, would eventually lead to a correct design satisfying all constraints. However, no support is provided at present for recovery from constraint inconsistency when constraint propagation is switched back on.

5.4. Constraint Editor

The constraint editor is a user interface for inspection and manipulation of constraints and variables. The editor provides a convenient user interface through which a user can walk through a network of constraints, and make any changes to the network which are necessary to remove a constraint violation. Besides being used as a debugger for constraint violation, constraint editors are useful for user entry of design specifications, and user inspection of existing design constraints. Fig. 5.4 shows a screendump of a constraint editor.

The editor provides special functions associated with manipulation of constraints and variables. The user can examine all variables associated with a constraint, or all constraints associated with a variable. The user can trace all variables which are consequences of a variable value or of a constraint and trace all variables which are antecedents of a variable value. The user can also instantiate or remove a constraint or a variable through the constraint editor. Finally, the user can make value assignments to variables through the constraint editor, turn off or on constraint propagation in the system, and restore all visited variables in the last propagation to their original states.

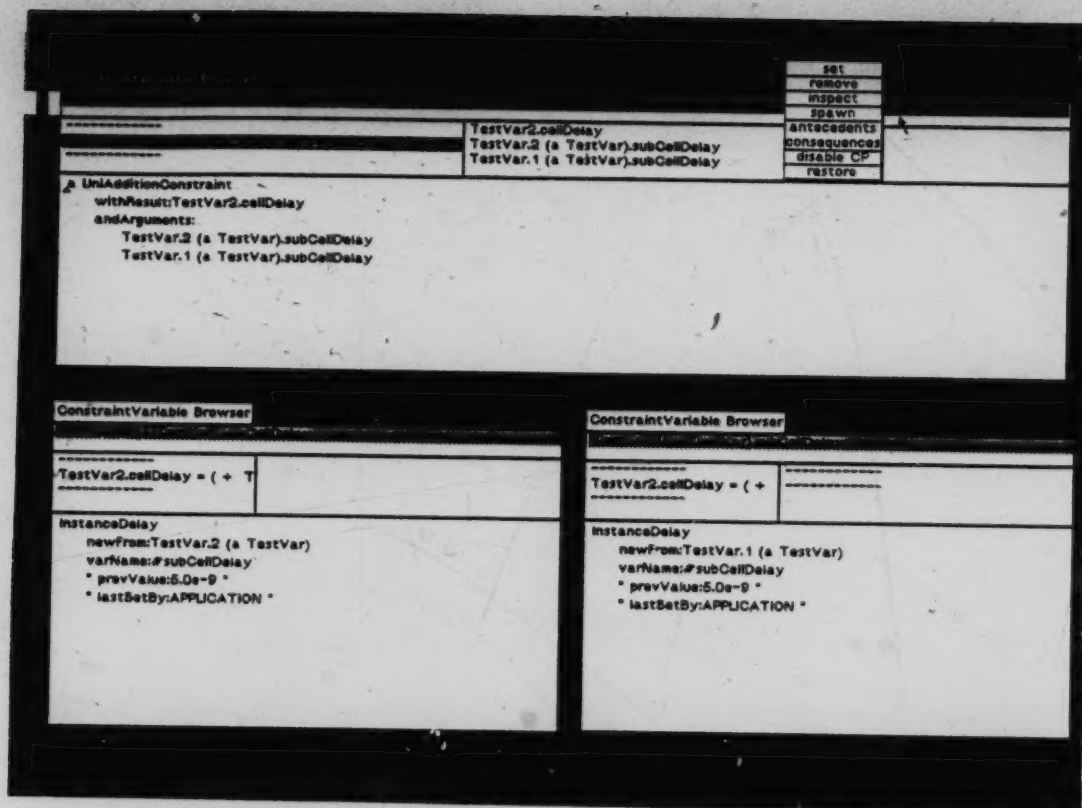


Figure 5.4 Screenshot of Constraint Editors

Chapter 6. Consistency Maintenance

Sample applications have been implemented to demonstrate the usefulness of constraint propagation in STEM. These are consistency maintenance for tool integration, incremental design checking for interactive designs, and module validation in automated module selection. This chapter presents STEM's approach to application interface and consistency maintenance in which constraint propagation is used to facilitate tool integration.³ The next two chapters present the second and third sample applications respectively.

6.1. Introduction

In an integrated design environment, multiple representations of a single design are required in order to interface the design database to a variety of tools. For example, while a layout tool views a design as a set of geometries, a schematic editor views the same design as a collection of transistors and nets. In order to integrate these tools in the design environment, both of these design representations must be accommodated by the environment. Consequently, when a design is modified, all representations of the design must be updated, otherwise inconsistency may exist among different design representations. Consistency maintenance in integrated environments refers to the data management of different design representations to prevent inconsistencies in the environment.

In STEM, consistency maintenance is achieved by the way application programs interface to the design database. The following sections present STEM's approach to application interface, and discusses the way consistency is main-

³ The content of this chapter has been published in [Ly87].

tained in the environment.

6.2. Application Interface: Related Work

Engineering environments which deal with multiple design representations by explicitly storing separate copies of the representations in the database allow inconsistencies of design data. For example, Palladio [Brow83] allows the definition of multiple perspectives of a cell. Perspectives are explicit data representations manipulated by different design tools. Perspectives are independent of one another, and different combinations of perspectives collectively define different designs. In general, information in different perspectives may overlap, resulting in the need to maintain consistency and to conserve database storage. Design validity in such systems is often enforced by executing consistency checking programs upon check-in of a design representation [Katz83]. This increases the complexity of the task of tool integration, because it is not always easy to check the consistency between two different representations of a design.

The Fred database [Wolf86], on the other hand, incorporates procedural knowledge to support engineering designs. Access functions (methods) in an object-oriented, hierarchical database implement algorithms that calculate properties of designs. Users may specify default values or approximation functions to speed up calculations. Fred also features "dependency", which executes a method whenever a particular message is sent to particular module(s). Calculation results apparently are not automatically stored in the database, so there are few consistency problems, although inconsistencies involving (user specified) default values can exist.

Wiederhold [Wied86] proposed the use of view-objects as an interface

between application programs and an engineering information system (EIS) database. View-objects are dynamically generated from the data in the engineering database. The view-object generators hide the implementation details (e.g. the relational organization) of the database, and generate objects as required by the applications. However, the interface does not maintain consistency among the database and view-objects in different applications.

6.3. STEM's Approach to Application Interface

In STEM, calculated views are used to interface application programs to the object-oriented database. The database objects contain only essential design information, from which different design representations are calculated and presented to applications through different views. Views implement abstractions of the design objects and are optimized for their corresponding applications. The derived data in views are erased (reset to NIL) whenever any data they depend on changes. Recalculation of erased data is delayed until the next time they are needed by applications or the designer. This maintains data consistency among views and the database without a severe penalty on database updates.

Application programs in STEM can be invoked explicitly through controllers by the user or implicitly by the database objects when the results of the applications are needed. When the user selects a menu item, for example, the associated controller invokes the corresponding application by sending a message to the model. Implicit invocation of applications is achieved by associating invocation messages with property variables (i.e., STEM variables whose values are calculated results of specific application programs). Property variables are implemented like special daemon objects (Fig. 6.1). When a property variable

is accessed (read), the variable object checks if its value is NIL. If it is, the associated invocation message is sent, invoking the appropriate application program which updates the value of the variable.

```
Variable subclass:#PropertyVariable
  InstanceVariableNames:'reCalculateMessage arguments evalFlag'
  classVariableNames:''
  poolDictionaries:'VisitedConstraintsAndVariables ConstraintScheduler'
  category:'Variable-Hierarchy'
```

PropertyVariable class methodFor:'instance creation'

```
newFrom:parent reCalculateMessage:symbol withArguments:args
  temp ← self new.
  temp parent:parent; reCalculateMessage:symbol; arguments:args.
  temp initialize
  ↑ temp
```

PropertyVariable methodFor:'evaluate'

```
value
  "evalFlag is essential in preventing infinite loops of evaluation calls"

  prevValue isNil
  ifTrue:[
    evalFlag isNil isTrue:[
      evalFlag ← #evaluating.
      self reCalculate.
      evalFlag ← nil]].

  ↑ prevValue
```

reCalculate

```
↑ self parent perform:reCalculateMessage withArguments:arguments asArray
```

Figure 6.1 Declaration and methods for PropertyVariable

Implicit invocation is combined with constraint propagation to form part of the consistency maintenance facility of the database. An "update-constraint" specifies dependencies of a (set of) property variable(s) on certain data. Whenever any of these data change, constraint propagation across the update-constraint erases (i.e., resets to NIL) all property variables in the constraint.

Implicit invocation then recalculates these erased property variables the next time they are accessed. This combination of constraint propagation and delayed recalculation ensures the internal data consistency of the database and reduces recalculation of data.

STEM's approach to tool integration improves upon past research in two ways. First, a view is dependent on a single database object (model) from which the view's data are calculated, so consistency among different representations can be maintained by recalculation of the view data whenever necessary. The results of database updates by one tool are immediately apparent to other tools, without a severe penalty on database updates. Second, tools are integrated into the database maintenance support by implicit invocation and constraint propagation. By embedding constraint propagation in the database maintenance mechanisms, changes in the database can be propagated along variable dependencies declared by tool designers and/or users, and trigger delayed recalculation of internal data to maintain consistency of the database.

6.4. Application Interface

There are two classes of application programs integrated into STEM. Internal application programs written within STEM are Smalltalk object-oriented programs that have direct access to objects in the environment. External application programs written independently of STEM are any UNIX executable files and do not have direct access to STEM objects.

6.4.1. Internal Application Programs

STEM builds on top of Smalltalk. Users can write applications in Smalltalk, which then become part of STEM. These internal applications have to be interfaced to the STEM design database. Each of these applications may require data structures other than those found in the database. Instead of modifying the application to use an inflexible database organization, views are used to customize the database to suit the application. Associated with each application is a view definition, which maps database objects to view objects, with internal variables and methods that facilitate the application. Views localize and hide implementation details of the database, and are shared among applications with similar data requirements. This use of views is illustrated by the four tile-based module compilers [Law85] implemented in STEM.

Instances of module compilers generate a compiled cell's internal structure based on the placement, orientation and size parameters specified in the compilers. A VectorCompiler builds a linear array of subcells (instances of cells), a WordCompiler builds a vector of subcells with special end-cells, and a MatrixCompiler generates a two-dimensional array of subcells. A GraphCompiler allows the user to graphically specify module builders that are able to generate more complicated structures [Mayo86].

When defining a compiled cell, the cell designer specifies the kind of module compiler to be used for the cell, and an instance of that compiler class is created and assigned to the cell as its structureLayout instance variable [Girc87]. The designer then proceeds to specify the placements and orientations of the compiler. All butting io-pins establish connections between their respective signals. However, the designer can disallow connections between two signals in a GraphCompiler, which withdraws the non-connecting io-pins

from the boundary of a cell. Fig. 6.2 is a screendump showing (from bottom-left to right) the specification of a 2-bit adder slice, its use in a 5-bit (GraphCompiler) adder, and the structure and layout of the compiled 5-bit adder.

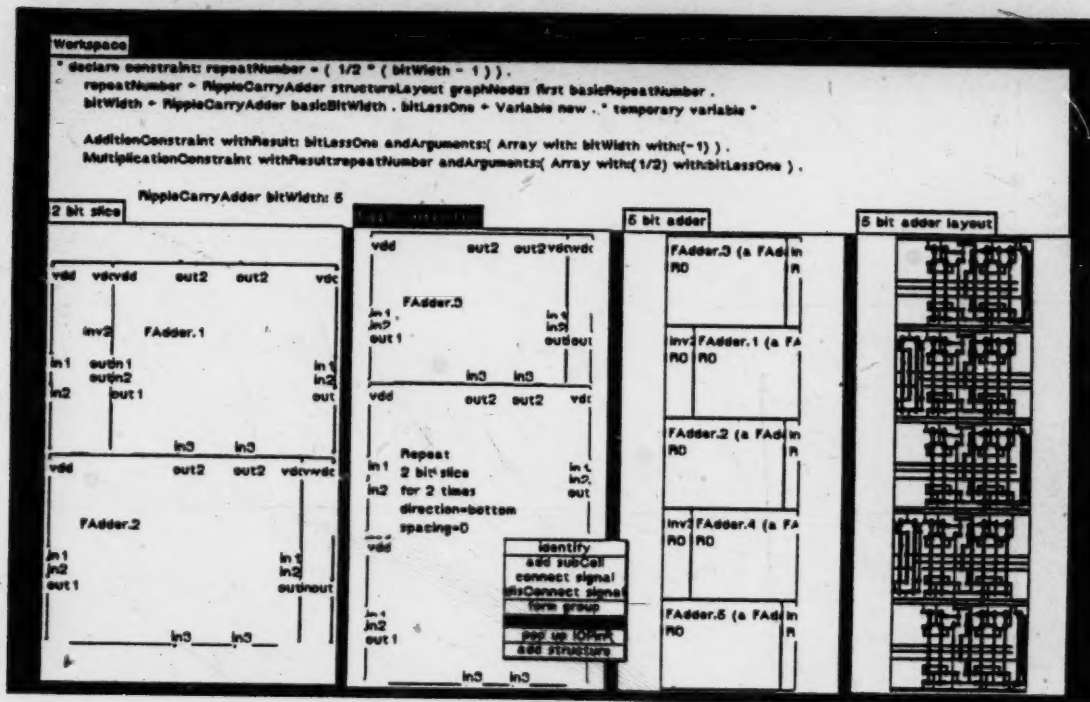


Figure 6.2 Building a 5-bit Adder with a GraphCompiler

The compilation routines treat subcells as black boxes, and instances of *CompilerView* are used to interface the actual subcells (in the database) to these routines. Only the bounding box and the io-pins of a subcell are visible through its compiler view. Moreover, the compiler views organize the io-pins of their models in four lists (top, bottom, left and right), sorted according to their locations (increasing x, y coordinates), to suit the access pattern of the compiler routines (the primary operation is butting io-pins of adjacent subcells). On the other hand, database subcells do not explicitly store any io-pin, and the data-

base cells store their io-pins unsorted and grouped according to their respective signals. Subcell io-pins are calculated from cell io-pins by applying the subcell's placement transformation to copies of the cell io-pins.

Without using compiler views, the module compilers would either trigger recalculation of io-pin transformations and sorting of io-pins on every subcell io-pin query, or store all transformed and sorted subcell io-pins as temporary variables inside a global compilation routine. The first approach sacrifices efficiency, and the second deviates from sound object-oriented programming. Using compiler views avoids unnecessary recalculations while retaining the logical integrity of the subcells. The compiler routines rotate, translate, and assign signal connectivities to instances of compiler views as if they were the actual subcells. The views in turn carry out these operations by sending their models the appropriate messages. Data in views are erased whenever their models change, and recalculation is triggered the next time the compilation routines access the views for data.

6.4.2. External Application Programs

External application programs differ from internal applications in that they interface to STEM through textual data-streams (UNIX files and pipes), and they have predefined input and output formats. This requires special internal routines for parsing and formatting of the input and output texts. An external application is integrated in STEM through a corresponding internal application which serves as an abstract model of the external process. The internal application is responsible for file-out (pipe-out) of formatted data, initiation of the external (background) process, and file-in (pipe-in) of the results. Using this approach, the external implementation of an application is hidden from the database, and

views still serve as the interface between the application and the database. The implementation of a SPICE interface illustrates this.

The internal interface to SPICE consists of three user interfaces: SpiceNet handles extraction and editing of SPICE net-lists, SpiceSimulation handles editing of simulation commands and viewing of SPICE output file, and SpicePlot handles graphical display and measurement of SPICE output waveforms. Figure 6.3 is a screendump showing (clockwise from the top-left corner) a cell of three cascaded inverters, its SpiceNet window, its SpiceSimulation window and the SpicePlot window of the simulation results.

SpiceNet maintains correspondence pointers between words in a SPICE net-list and the actual subcells and nets, abstracting a database cell into a paragraph of text. It implements a view through which a text editor can manipulate the database object, allowing the user to edit the database cell through editing its SPICE net-list. A SpiceNet is updated whenever its model (a cell) changes. When the user selects the "simulate spice" menu option, a SpiceSimulation user interface opens up on the screen.

The SpiceSimulation user interface is a restricted text editor. It contains a text view of a database cell (i.e., the expanded SPICE net-list of the cell), and (default) simulation parameters and options. The extracted net-list, displayed in normal font in the SpiceSimulation window, is not editable. Editable text, displayed in bold in the window, includes simulation circuit, parameters, commands and options. When the user selects the "run spice" menu option, the simulation text is filed out, and the SPICE process is run in the background. Control is returned to the user as soon as the external process is initiated. When SPICE terminates, its output is filed in and listed in the simulation output window, and the user is informed of the termination. The user then has the

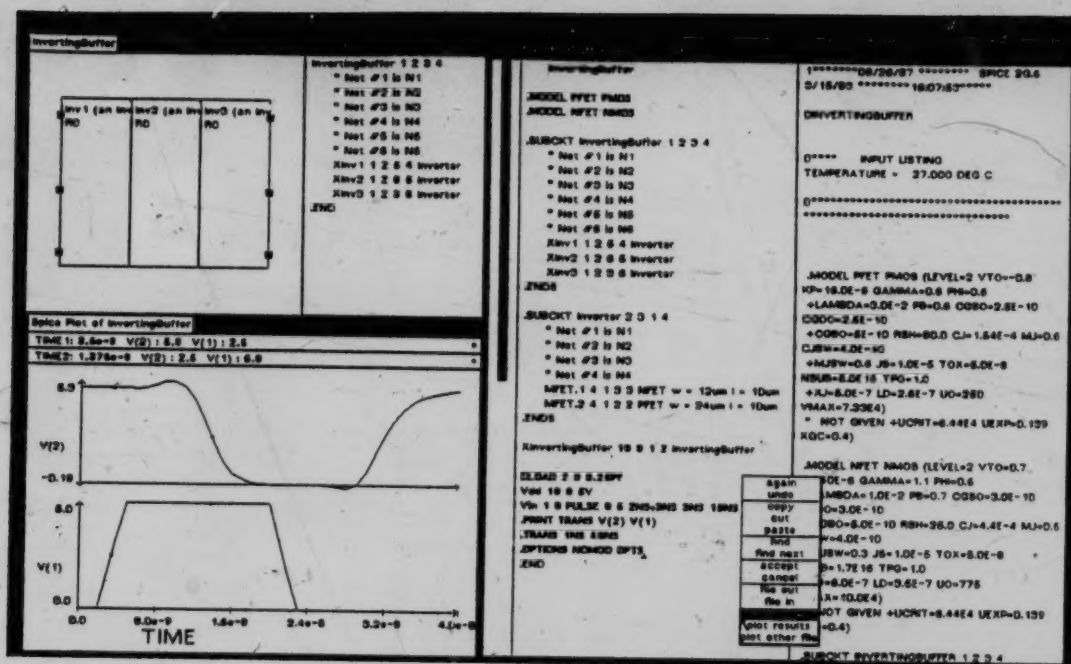


Figure 6.3 SpiceNet, SpiceSimulation and SpicePlot

option of plotting the output waveforms using a SpicePlot, which allows a host of graphical manipulations (display options and point-to-point measurements) on the waveforms.

SpicePlots are associated with the SpiceSimulations from which their waveforms are generated. This allows comparison among plots from different simulation parameters. In order to avoid misleading the user, all SpiceSimulation and SpicePlot windows on a cell are marked outdated (in the label field of the windows) when the cell's net-list is changed.

6.5. Change Management

Change management in STEM involves the maintenance of both internal and external consistencies. Internal consistency (consistency among related variables within the design database) is maintained by delayed updates of related variables whenever variables are changed in the database. External consistency (consistency among calculated views and the design database) is maintained by delayed updates of relevant data in the views whenever a database object changes.

6.5.1. Internal Consistency Maintenance

In order to reduce the overhead of unnecessary calculations, derived data that are expensive to compute or frequently accessed are stored with the database objects as collections of property variables (storedViews). Associated with each property variable is a message which, when sent to the database object, invokes a tool that calculates the current value of the property variable. The tool is invoked whenever the property variable is queried and is NIL. This is achieved simply with a condition check in the variable's query method. Property variables are reset to NIL when they are out of date by update-constraints which specify validity dependencies of property variables on other variables. Update-constraints are declared by tool designers when integrating the tools into the database.

Update-constraint and implicit invocation provide a convenient way in which new tools can be easily integrated into STEM without disrupting the consistency of the environment. Instead of modifying many access routines in a cell to ensure data consistency when introducing a new tool, tool integrators can specify a set of equivalent update-constraints in the initialization routines of the

cell. This simplifies tool integration, localizes dependency information, and facilitates maintenance of tools. Since update-constraints are coded in the initialization routines, inheritance of these constraints is simply achieved with Smalltalk's inheritance of the initialization routines. Update-constraints are instantiated whenever a cell (instance or class) is initialized, and discarded only when the cell is discarded from the system.

Other than update-constraints, many different types of constraints may be instantiated in the design process to maintain arbitrary relationships among variables and parameters in a cell. This use of constraint propagation maintains dependencies among localized groups of variables, provides documentation for design intentions, and reduces the amount of data the designers must specify. For example, if a designer intends to maintain an equality relationship between two parameter values for all instances of a cell, then an equality constraint on these parameter variables can be created for each cell instance during its initialization (Fig. 6.4). This constraint then maintains the equality relationship by updating the value of one parameter whenever the other parameter changes value, and vice versa. Moreover, this constraint documents the equality between these two parameters as a design intention, as opposed to an incidental design characteristic.

```
GraphCompilerTestCell methodFor:'initialize'

Initialize
  super initialize.
  EqualityConstraint with:(self instVarNamed:#p1)
                    with:(self instVarNamed:#p2).
  ↑ self
```

Figure 6.4 User-specified EqualityConstraint in initialization method

6.5.2. External Consistency Maintenance

Views are dependents of their models. Whenever an object changes a database object (a model), it must send the database object the message "#changed". The model in turn broadcasts the same message to its dependent views to inform them of the change. The views respond to the message by erasing their calculated data. Selective erasure of views can be achieved by the alternate message, "#changed:key", where key is a symbol indicating the nature of the changes. For example, the key "layout" signifies that only the layout of a model is changed and no electrical connectivity has been modified. Thus, a SpiceNet view of the model does not have to erase its data (unless it contains extracted parasitics).

Besides its dependent views, a changed cell also broadcasts the changes to those cells containing it (i.e., cells containing instances of the changed cell). This propagation of changes up the design hierarchy terminates at those cells whose external properties (e.g., bounding box, io-pin locations, io-signal types) are not affected by the change. This maintains the external consistency of the database.

The mechanism of change broadcast is basically the procedural equivalent of the update-constraint and its associated propagation mechanism. The reason for the procedural implementation is that, since views are external to the database, view designers should not have to know the internal variables of the database objects. Also, the dependencies among views and models are sufficiently well defined that the propagation can be proceduralized for security and performance. This exemplifies a tradeoff between flexibility and explicitness of declarative constraints and the efficiency and rigidity of procedural constraints.

6.6. Summary

In this chapter, the way application programs are integrated in STEM has been described. Calculated views are used to implement a data interface between applications and the object-oriented database. The derived data in views are dependent on their models, and are erased whenever necessary to maintain external data consistency of the database. Applications can be explicitly invoked through the controllers, or implicitly invoked through property variables. Update-constraints are used to specify dependency among calculated variables and other data, and constraint propagation removes potential inconsistencies by erasing property variables whenever the data they depend on changes. Implicit invocation and constraint propagation integrate tools into the environment as part of the consistency maintenance facility of the database.

Chapter 7. Incremental Design Checking

The second sample application of constraint propagation is incremental checking in interactive design. Design checking verifies the correctness of a design. This involves extracting design characteristics and comparing them against circuit specifications and design constraints. Traditional design checking programs are run in batch mode over large portions of a design at a time. This postpones detection of design errors, often requiring major redesign to undo consequences of these errors. However, with integrated environments, design checking programs can perform incremental checking. For example, layout systems like Magic [Oust85] incorporate incremental layout design rule checkers which warn designers about design rule violations as the layout is edited. By checking only those portions of the design which have changed, incremental design checking can achieve fast enough response times to be run concurrently with design editing. This gives designers immediate feed back on design errors and helps reduce major redesigns and futile design explorations in blind alleys.

Incremental checking of design entry usually requires the maintenance of a global data structure and a program that maintains this structure. Instead of using a number of different data structures and programs that implement different kinds of design checking, STEM uses constraint networks and constraint propagation to implement incremental checking of designs. Constraints capture both design specifications and dependencies of design characteristics. Propagation of constraints provides the engine for incremental checking of specifications as well as incremental derivation of design characteristics. Since constraint propagation is hierarchical in STEM, design characteristics in low levels of the design hierarchy can be propagated up the hierarchy and checked

against design specifications at higher levels. Furthermore, constraint propagation implements an open-ended framework for incremental design checking: arbitrary design checking can be added to the system by introducing additional types of constraints.

Incremental checking for signal types, bounding box and delays of designs have been integrated into STEM. These constraints may be instantiated explicitly by designers, or implicitly by STEM as designs are entered. For example, the designer can specify the maximum delay between two signals of a cell with a "less-than" constraint on the appropriate delay variable. When the delay value is eventually calculated, it is checked against this user-specified constraint. On the other hand, whenever two signals are connected by a net, STEM adds appropriate signal typing constraints which ensure that connected signals have compatible types and equal bit-widths. Whenever a signal type is specified by the designer, the signal types of other unspecified signals on the same net are inferred and propagated. This reduces the amount of data entry and facilitates design changes.

The following sections discuss the instantiation, propagation and testing of the different types of constraints used to check signal types, bounding box and delays of designs.

7.1. Signal Types

Each io-signal and net in STEM has three properties: bitWidth, dataType and electricalType. The bitWidth of a signal is simply the number of bits of the signal. The dataType of a signal indicates the type of data (e.g., integer, boolean, real) that are carried by the signal. The electricalType of a signal indicates the electrical properties of the signal (e.g., DIGITAL, ANALOG, TTL).

Signal typing constraints are implied by nets. A net electrically connects signal variables in subcells to one another, and possibly to signal variables in the cell containing these subcells. The signal types of a net depend on those of the signals connected to that net. Signal typing constraints check for incorrect connectivities among signals, prevent improper uses of cells and infer unspecified signal types from signal connections. Three kinds of signal typing constraints currently implemented are equality-constraint on bitWidths, and compatible-constraints on dataTypes and electricalTypes.

Equality-constraints on bitWidths simply check that bit widths of connected signal variables are the same. The propagation method of an equality-constraint tries to set all of its argument variables to the same value as that of the variable which changed value. A propagated bitWidth value is rejected by a signal variable if the signal has a constrained bitWidth that has a different value than the propagated value. A constrained bit-width usually results from realizations of other nets connected to the signal, or from designer's specification of the signal's bit width. For example (Fig. 7.1), if the internal structure of a cell connects an 8-bit net to an input signal, then this signal is constrained to be 8 bits wide. Alternately, the designer can assign a bit width of eight to this input signal before designing the internal structure of the cell. In either case, when a 4-bit net is connected to the corresponding signal variable in an instance of the cell, constraint violation warns the designer of the incompatible bitWidths.

Compatible-constraints check that their argument variables are compatible. For each net, there are two compatible-constraints relating all signals connected by the net: one for the dataType variables of these signals, another for the electricalType variables of these signals. The data and electrical types of signals are defined hierarchically, with the most abstract types at the roots of the hierar-

chies and least abstract types at the leaves of the hierarchies. These are simply implemented with Smalltalk's class hierarchy (Fig. 7.2).

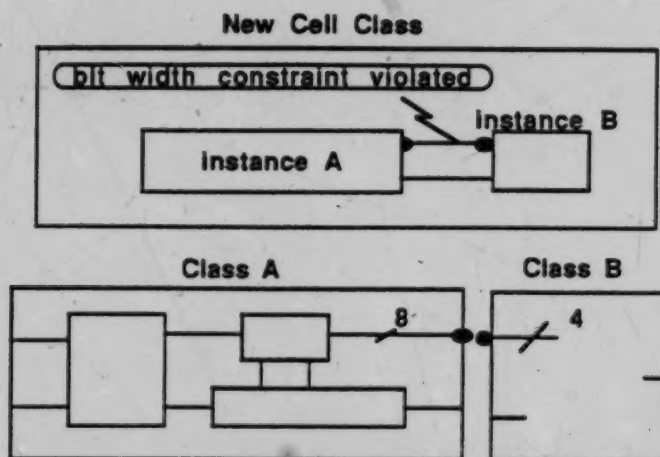


Figure 7.1 Example of bit width constraint violation

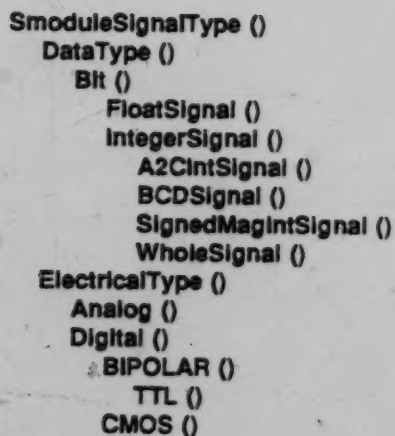


Figure 7.2 Signal type hierarchies

Compatibility of signal types is defined by their relative positions in the type hierarchy. Basically, two types are compatible if and only if one is a sub-type of

the other. Provided that two types are compatible, one type is more abstract than another if its type is an ancestor of the second type. These tests are easily implemented using Smalltalk's method for subclass enumeration (Fig. 7.3).

`SmoduleSignalType class methodsFor:'test'`

`isCompatibleWith:anObject`

```
"I am compatible with anObject if I am a subclass of anObject
or anObject is a subclass of me"
self == anObject ifTrue:[ ↑ true ].
(self allSubclasses includes:anObject) ifTrue:[ ↑ true ].
(anObject allSubclasses includes:self) ifTrue:[ ↑ true ].
↑ false.
```

`isLessAbstractThan:anObject`

```
"I am more abstract than anObject iff anObject is a descendent of me"
↑ self allSubclasses includes:anObject
```

Figure 7.3 Type hierarchy test methods

Each compatible-constraint has two instance variables: "netVariable" and "arguments". The first variable contains the type variable of a net and the second variable is a list containing the type variables of all signals connected by the net. Assuming that all signals in a net are compatible, the signal type of the net is then the least abstract type of all signals in the net. Similarly, provided that all signals connected to an untyped signal are compatible, the signal type of the signal is then the least abstract type of all of these signals. This is achieved with the overwrite rule of propagated values in the typing variables: a propagated type value can only overwrite a previously propagated value if the new value is the less abstract type of the two (Fig. 7.4).

Every time a signal is added to a net, signal typing constraints of the net are updated. The signal's bitWidth variable is added to the equality-constraint of the net's bitWidth variable. Similarly, the signal's dataType and

SignalVariable methodFor:'assignment'

```
setTo:anObject constraint:aConstraint justification:justification
(self canChangeValueTo:anObject)
  ifTrue:[ ↑ super setTo:anObject
           constraint:aConstraint
           justification:justification ]
```

SignalVariable methodFor:'test'

```
canChangeValueTo:anObject
  "I can change value to or from NIL freely"
  (prevValue isNil or: anObject isNil) ifTrue:[ ↑ true ].
  "otherwise, can only change to more abstract value"
  ↑ prevValue isLessAbstractThan:anObject
```

Figure 7.4 Code segments showing signal variable overwrite rule

electricalType variables are also added to the compatible-constraints of the net's dataType and electricalType variables, respectively. On the other hand, every time a signal is disconnected from a net, the signal's bitWidth, dataType and electricalType are removed from the appropriate constraints of the net's variables.

The data type and electrical type of a signal are properties of the cell class and do not vary among different cell instances. As a result, the dataType and electricalType variables of a cell instance's signals are those of the corresponding io-signals in the cell class. This is also true for bitWidth variables of signals in composite cells where signal bit widths are the same for all instances of a cell. As a result, a type variable for a cell class signal can have a large number of signal typing constraints, each due to a net connected to the corresponding signal in a cell instance and representing requirements imposed by a different environment in which the cell is used. In Fig. 7.5, a signal type variable in class A is constrained to be compatible (indicated by constraint "C") with signal types in class C and B through two different instances of A.

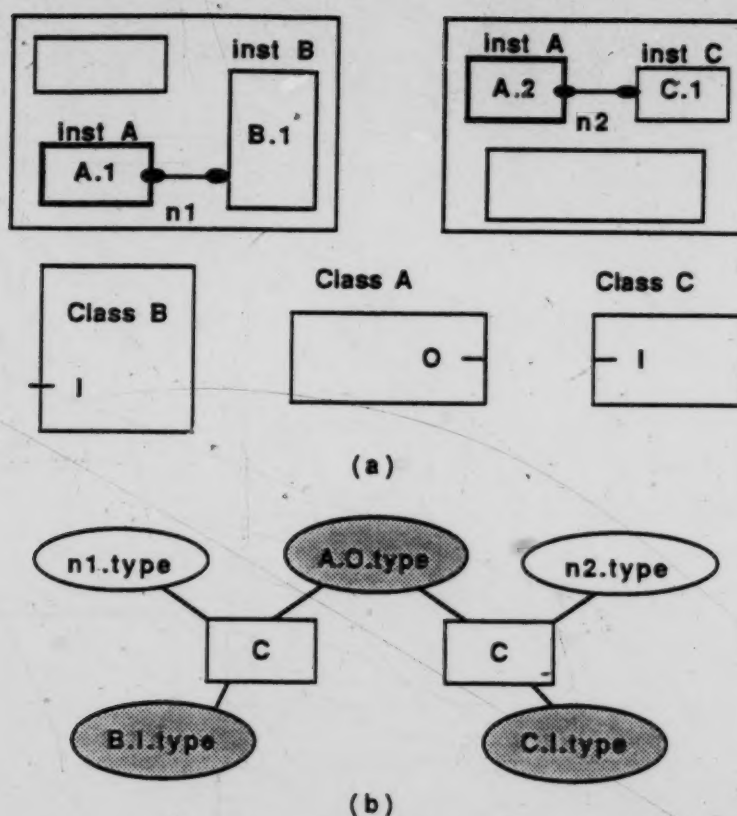


Figure 7.5 Signal type constraints due to different cell instances

Compiled cell instances of the same class may have different bit widths, so signals for these cell instances may have their own bitWidth variables. However, dataType and electricalType variables remain properties of signals in cell classes.

With explicit constraints, incremental checking of signal types can be performed whenever a net is connected to or disconnected from a signal, and whenever a new signal type is assigned to a signal. The type specifications of a cell's signals can be incrementally refined by different uses (instances) of the

cell, and can subsequently be used to guide the internal design of the cell.

7.2. Bounding Box

The bounding box of a cell class is the smallest rectangular box that contains the internal structure of the cell. The bounding box of a cell instance is a rectangular area in which the cell instance is placed. While a cell instance cannot be placed in an area smaller than its class bounding box, it can be placed in an area equal to or larger than its class bounding box. The mapping of a cell's internal structure into the bounding box area of a cell instance is described by the transformation matrix of the cell instance. For cell instances whose bounding boxes are larger than those of their class bounding boxes, STEM provides stretching routines that extend signal ports to the perimeter of the bounding box (Fig. 7.6).

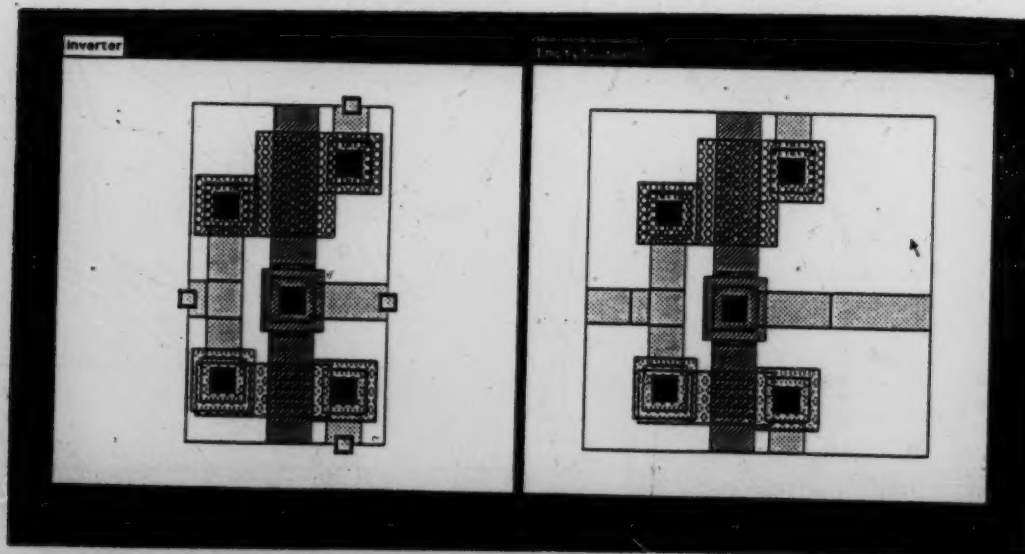


Figure 7.6 Screenshot showing stretching of io-pins to bounding box

Bounding boxes of cell classes are implemented with "ClassBBox" variables, while those of cell instances are implemented with "InstanceBBox" variables. Whenever a bounding box of a cell class is assigned a new value, it propagates this new value to the bounding box of all instances of the cell. This defaults all instance bounding boxes to be the same as that of the new class bounding box (with appropriate transformations), and checks that no instance bounding box is larger than the class bounding box. On the other hand, when an instance bounding box is assigned a new value, it is checked with the corresponding class bounding box to make sure that it is not smaller than the class bounding box. However, no propagation is performed from instance bounding boxes to class bounding box (Fig. 7.7).

InstanceBBox methodFor:'test'

ImmediateInferenceByChanging:aClassBBox

"If I am nil, or a propagated value from aClassBBox,
then update myself"

(self value notNil and: [self lastSetBy key = #USER]) ifTrue: [↑ self].

self setTo:(self parent sTransform applyTo:aClassBBox parent boundingBox)
constraint:aClassBBox
justification:nil

ClassBBox methodFor:'test'

IsSatisfiedBy:InstanceBBox

"returns true iff instanceBBox can contain my transformed value"

bBox ← instanceBBox value.

bBox isNil ifTrue: [↑ true].

selfBBox ← instanceBBox parent sTransform applyTo:self parent boundingBox.

↑ bBox extent > selfBBox extent

Figure 7.7 Code segments for bounding box

The minimum bounding box of a cell class is a function of the internal structure of the cell. An inherited cell class routine, "calculateBoundingBox", calculates the class bounding box from the nets and subcells in the cell.

Whenever a subcell changes its bounding box ("InstanceBBox"), the class bounding box has to be updated. There is therefore an update-constraint from bounding box variables of all subcells and nets to the bounding box variable of the cell classes containing them. Instead of maintaining one update-constraint for every instance of "ClassBBox" in the system, STEM implements the update of class bounding boxes procedurally. When an "InstanceBBox" is changed, it resets the bounding box variable of the parent cell of its cell instance (Fig. 7.8). Similarly, when a net changes its connections, it resets the bounding box of its parent cell. The procedural implementation of this update-constraint is warranted because the operations are localized, well defined and are performed very frequently. The flexibility of a declarative constraint is not needed.

InstanceBBox methodsFor:'assignment'

setTo:anObject justification:justification

```
...
self parent parentCell notNil
  ifTrue:[
    (self parent parentCell parent instVarNamed:#boundingBox)
      setTo:nil justification:#UPDATE ].
...
```

setTo:anObject constraint:aConstraint justification:justification

```
...
self parent parentCell notNil
  ifTrue:[
    (self parent parentCell parent instVarNamed:#boundingBox)
      setTo:nil justification:#UPDATE ].
...
```

Figure 7.8 Subcell bounding box updates parent bounding box

Other constraints on bounding box variables can be declared by designers. Area, pitch-matching and aspect ratio constraints can be specified. Fig. 7.9 shows the definitions required to define a new aspect ratio constraint type to the system. Once this is defined, user can constrain any boundingBox variable to a

specified aspect ratio by instances of `AspectRatioPredicate`. Whenever a cell changes its bounding box, constraint propagation adjusts default bounding boxes of cell instances, checks if any instance bounding boxes are too small for their class bounding box, and checks if other, user-specified constraints are violated.

```
PredicateConstraint subclass:#AspectRatioPredicate
instanceVariableNames: 'xYRatio'
classVariableNames: ''
poolDictionaries: 'VisitedConstraintsAndVariables' 'ConstraintScheduler'
category: 'Constraint-Predicates'
```

`AspectRatioPredicate` methodFor: 'test'

```
isSatisfied
self arguments do: [ :arg |
  arg value isNil ifFalse: [
    (arg value x) / (arg value y) = xYRatio
    ifFalse: [ ↑ false ] ] ].
↑ true
```

`AspectRatioPredicate` class methodFor: 'instance creation'

```
xYRatio:value andBoxVar:var
| temp |
temp ← self new.
temp xYRatio:value.
temp addArguments:(OrderedCollection with:var).
↑ temp
```

Figure 7.9 `AspectRatioPredicate` Definition

7.3. Delay

Delay constraints incrementally compute the worst case delay estimates between input and output signals of cells by searching for the longest paths in the delay networks. A simple delay model similar to that of CRYSTAL [Oust83] is used (Fig. 7.10), and the assumption is made that delays of cascaded components are additive.

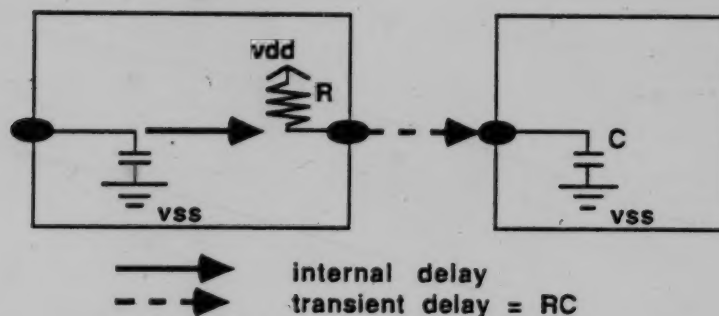


Figure 7.10 The delay model

The RC model is used for adjusting delay estimates with respect to the output resistance of output signals and the loading capacitance of input signals. Default values for delay variables can be specified to speed up delay calculation. Different delay models and algorithms can be specified in subclasses of delay constraints and variables to overwrite the default model and algorithm. While delay constraints provide quick estimates of signal delays based on nominal delay characteristics of component cells, accurate timing measurements must still be obtained through simulation with external applications like SPICE [Sale84] and SPLICE [VICP74].

There are two types of delay variables: "ClassDelay" and "InstanceDelay".

Instances of "ClassDelay" are used as delay variables between io-signals of cell classes, while those of "InstanceDelay" are used as delay variables between signals of cell instances. For each delay from an input signal to an output signal in a cell class, there is a corresponding instance delay variable from the corresponding input to the corresponding output in each instance of the cell (Fig. 7.11).

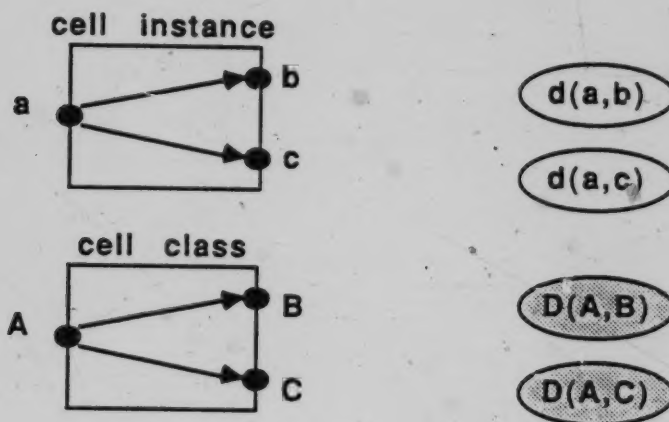


Figure 7.11 Dual delay variables in cell class and instances

When the value of a class delay variable changes, it propagates the new value to all of its corresponding instance delay variables, each of which adjusts its value according to the new cell delay value, the output resistance on its input net, and the total load capacitance on its output net. When these instance delay variables change their values, they trigger recalculation of all delay paths containing any of them. If such a delay path happens to be a critical path in a cell class, then a new value propagates to a class delay variables, and triggers further delay value adjustments. Delay characteristics are therefore incrementally calculated, propagated up the design hierarchy as soon as they are available, and checked with delay specifications at every level in the hierarchy.

Instances of "UniMaximumConstraint" and "UniAdditionConstraint" form the constraint networks that relate class delay variables of a cell to instance delay variables of subcells in the cell. An UniAdditionConstraint is a functional constraint which computes the sum of its arguments and is used to compute the total delay of a delay path from all component instance delays of the path. An UniMaximumConstraint is a functional constraint which computes the maximum of its arguments and is used to compute the longest delay of a class delay variable from all delay paths connecting the input and output signals of the class delay. These constraints implement delay networks that incrementally compute the delay characteristics of a cell from those of its subcells in STEM.

For example, Fig. 7.12(a) shows the structures and delay paths of two cells X_r and A. The corresponding delay constraint networks are shown in Fig. 7.12(b). The class delay $A.D(X,Y)$ implicitly propagates to its dual variables $A.2.d(x,y)$ and $A.1.d(x,y)$.

When input and output signals of a cell have been defined, the designer can specify the critical class delay variables of the cell, declare value constraints on these variables, and specify estimated delay values. Thereafter, other cell classes containing instances of this cell would be able to construct delay networks containing instance delays that correspond to these class delays. At this point, delay values computed by these constraint networks are based on user-specified delay estimates in this cell. After entering the internal structure of this cell, the designer can remove these delay estimates. STEM then constructs delay networks for this cell, calculates the values for class delay variables in the cell, and propagates these delay values to other cells containing instances of this cell.

Delay networks are constructed by enumerating all delay paths connecting

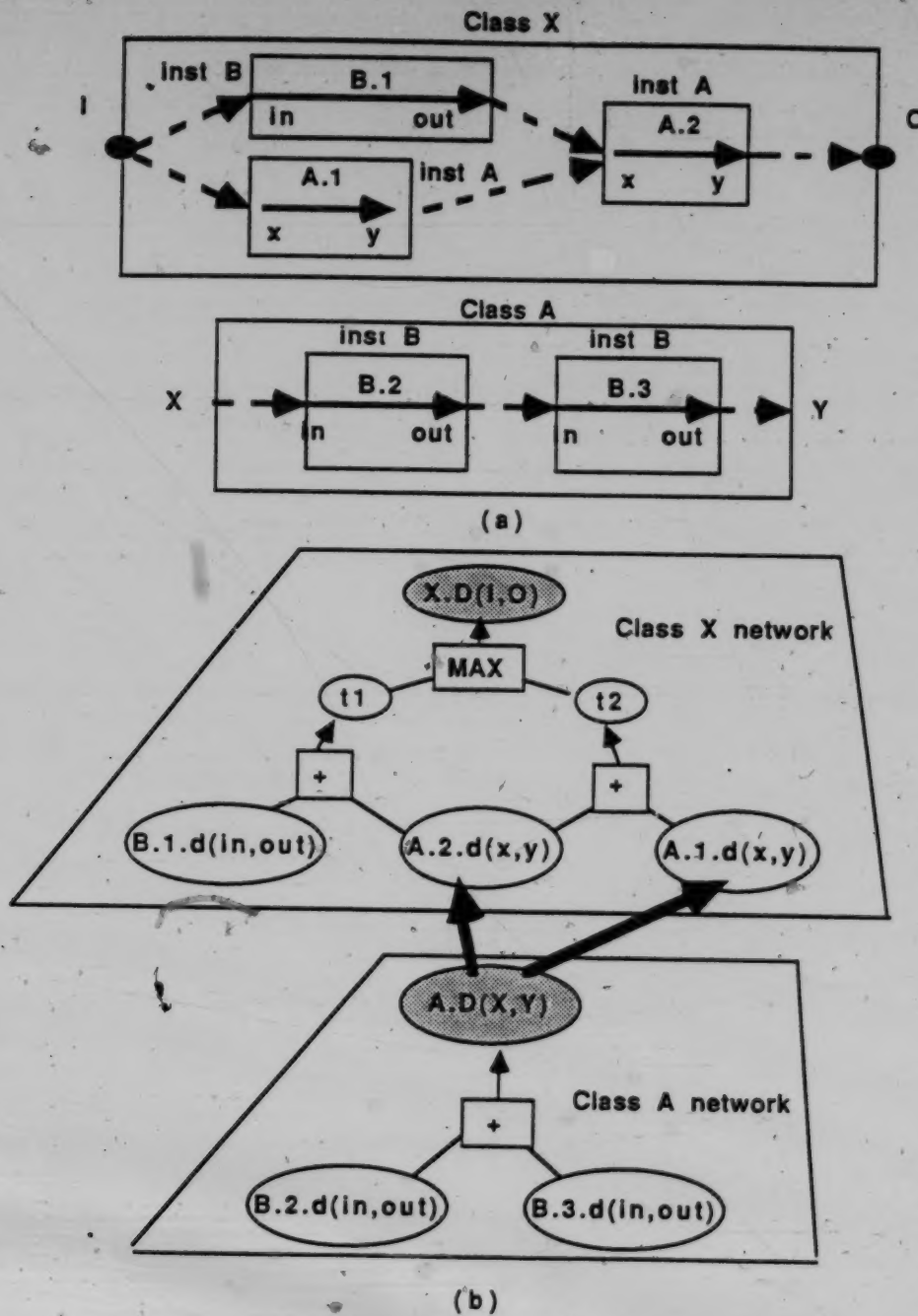


Figure 7.12 Hierarchical delay paths and delay constraints

the source and destination signals for each class delay variable in a cell, and then instantiating constraints that equate each class delay variable as the maximum of sums of instance delay variables in the delay paths. The delay path generation routines only consider those subcell delays that have corresponding user-specified delay variables in their cell classes. This gives cell designers the ability to focus STEM's attention to the critical delay paths in cells and reduces the extent of combinatorial explosion in delay path generation.

In order to maintain consistency, the delay networks of a cell are erased whenever the internal structure of the cell changes, and recalculated only when delay values are requested. Incremental editing of delay networks is not implemented due to efficiency considerations. Propagation of delay constraints is most useful in top-down designs, in which a cell is designed using subcells whose internal structures are yet to be designed. The incremental evaluation and checking of delay characteristics provide feedback on the impact of different internal designs of a cell on the delay characteristics of other cells containing this cell, and guide designers in exploration of the design space.

7.4. Summary

In this chapter, the way STEM implements incremental design checking using constraint propagation has been described. Constraints capture design specifications as well as dependencies of design properties so that design characteristics can be derived and checked incrementally. Hierarchical constraint propagation enables the system to check a design in the environments in which it is used. Incremental design checking provides an early feedback of design errors and helps prevent major redesigns.

Constraint propagation implements a very extensible framework for incre-

mental design checking. Additional design checks can be easily integrated into the existing system with new types of constraints. Designers can specify default values for design properties, and overwrite default algorithms and models with different property estimators in subclasses of constraints or variables.

The speed performance of constraint propagation is acceptable for interactive design checking provided that the constraint networks are sparse. Low-level design checks, such as layout design rule checking, are not suitable candidate applications for this approach because more specialized data structures (e.g., corner stitching) and constraint satisfaction algorithms (e.g., shortest-path algorithms on graphs) are necessary to achieve adequate speed and storage performance. However, higher-level constraint networks (e.g., delay, bounding box and signal types) tend to be sparse, and as such are good candidates for incremental design checking based on constraint propagation.

Chapter 8. Module Validation

In STEM, similar cells can be made subclasses of a "generic cell". Common properties of these cells can then be specified in the generic cell, and inherited by all descendent classes of the cell. Generic cells do not have physical realizations. However, they can be used in other designs in much the same ways as any other cell. The use of generic cells in a design facilitates design space exploration. The designer can specify generic cells with partial default characteristics for parts of a design and concentrate on the more critical parts of the design. Generic cells provide the general characteristics of their descendent cells and allow partial design checking before deciding on specific realizations. By deferring implementation decisions, generic cells allow a least commitment strategy of design. For example, a designer can instantiate a generic adder cell in a circuit, and decide on a specific realization (e.g., a ripple-carry adder) of this adder later. In the mean time, the rest of the circuit can be designed and checked against general characteristics of an adder. Subsequently, a suitable implementation of the adder can be selected to suit its surrounding circuitry.

Module selection is the task of selecting a valid realization of a generic cell instance in the context of a larger design. Automated module selection is an important step towards macro-cell designs that adapt to varying constraints in different contexts. Fig 8.1 shows a simple example of module selection. ADD8 is a generic, 8-bit adder cell with two subclasses: ADD8.RC, which uses a ripple-carry scheme, and ADD8.CS, which uses a carry-select scheme (Fig 8.1 (a)). ALU is designed with an instance of LU8, a logic unit of 8-bit, and an instance of ADD8, the generic adder. Given two different sets of design constraints of ALU, one with a tight area specification (Fig 8.1(b)) and the other

with a tight delay specification (Fig 8.1(c)), two different ALU implementations could result from module selection.

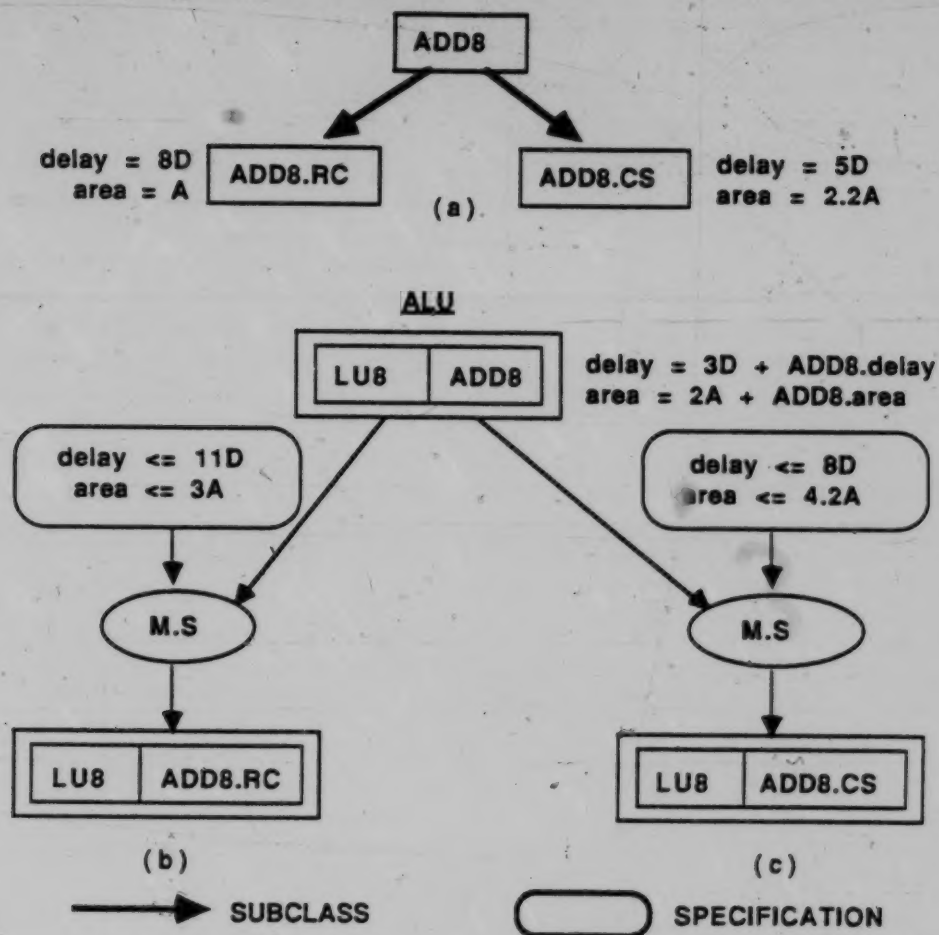


Fig. 8.1: (a) ADD8 has two subclasses;
 (b) tight area spec: use ADD8.RC;
 (c) tight delay spec: use ADD8.CS

A simple module selection routine has been implemented in STEM. It is based on the "generate and test" search algorithm and depends on constraint propagation to check for valid realizations of a generic cell. In the following sections, the algorithm, implementation and examples of module selection are presented.

8.1. Module Selection by Generate and Test

The "generate and test" search algorithm, augmented with selective testing and tree pruning, is used in module selection. The algorithm successively tests candidate realizations of the generic cell until all valid realizations are found. This algorithm is easy to implement for module selection in STEM because the candidate realizations of a generic cell consists of all descendent classes of the generic cell. Constraint propagation provides the test for feasible realizations of a generic cell instance. Since hierarchical constraint propagation is used as the test, the validity of a candidate realization depends on the environment of the generic cell instance.

A depth first traversal of the class hierarchy rooted at the generic cell class enumerates all candidate cell classes. For each of these candidate cells, tests are performed to determine its validity as the realization for the generic cell instance. The tests consist of setting different types of variables of the generic cell instance to values consistent with those for an instance of the candidate cell and checking for constraint violations. A valid realization is a cell whose instance variable values can be assigned to variables of the generic cell instance without violating constraints.

Module selection is implemented as a menu action in the Cell Browser [Girc87]. The user can select a generic cell instance in a cell, and invoke module selection through the menu. A list of all cell classes that can realize this generic cell instance is returned. However, no automatic replacement of the cell instance is attempted.

8.2. Techniques for Efficiency Improvement

Two techniques are used to improve the efficiency of generate and test, namely selective testing and tree pruning. Selective testing involves propagating and testing for selective types of property variables, starting with the most critical type. Presently, three types of properties are tested: delay, bounding box and signal types. Users can specify an ordered subset of these three types of properties and restrict the testing to the selected types and in the specified sequence (Fig. 8.2). For example, if the bounding box is the most constrained property in a design, followed by delay, and if signal compatibility is guaranteed (e.g., by the fact that no subclasses redefine signal properties), then the user can specify `##bBox #delays` to module selection when prompted for selective properties to test. Using selective testing, the designer can direct the system's attention to focus on only the relevant properties, and to apply the more critical tests first. This decreases the amount of constraint propagation for unsuccessful realization candidates, thereby improving the efficiency of the search.

CellWithVarAndConstraints class methodsFor: 'testing'

isValidRealizationFor: anInstance priorities: priorityList

```
priorityList do: [ :symbol |
    symbol = #bBox
        ifTrue: [ (self validBBoxFor: anInstance) ifFalse: [ ↑ false ] ].
    symbol = #signals
        ifTrue: [ (self validSignalsFor: anInstance) ifFalse: [ ↑ false ] ].
    symbol = #delays
        ifTrue: [ (self validDelaysFor: anInstance) ifFalse: [ ↑ false ] ].
    ↑ true
```

validBBoxFor: anInstance

```
[ bBoxVar v |
    bBoxVar ← anInstance boundingBoxVar.
    v ← bBoxVar prevValue.
    v isNil
        ifTrue: [
            "If instanceBBox is nil, then check if it can be set to my default"
            v ← anInstance sTransform applyTo: self boundingBox.
            ↑ bBox canBeSetTo: v ]
```

```

ifFalse:
    ↑ self boundingBoxVar isSatisfiedBy: bBoxVar ]

validDelaysFor: anInstance
| s d newDelay |
anInstance delayVariables do: [ :var |
    s ← var source sName asSymbol.
    d ← var destination sName asSymbol.
    newDelay ← self delayFrom: (self instVarNamed: s)
                to: (self instVarNamed: d)
                outputNets: OrderedCollection with: var destination net).
    (var canBeSetTo: newDelay) ifFalse: [ ↑ false ].
    ↑ true

validSignalsFor: anInstance
| mySignals bitWidthVar net |
" test signal dataTypes, electricalTypes, and bitWidths"
mySignals ← self allSignals.
anInstance allSignals do: [ :subCellSignalVar |
    bitWidthVar ← subCellSignalVar bitWidthVar.
    net ← subCellSignalVar net.
    net isNil & (bitWidthVar isKindOf: ClassBWidth) ifFalse:
        mySignals do: [ :ioSignal |
            ioSignal sName asSymbol = subCellSignalVar varName asSymbol
            ifTrue: [
                "If instance has user assigned bitWidth, check with my bitWidth"
                ((bitWidth isKindOf: InstanceBWidth)
                 and: [(ioSignal bitWidthVar isSatisfiedBy: bitWidthVar) not])
                ifTrue: [ ↑ false ].
                "If net is not NIL, check signal types with net"
                net notNil ifTrue: [
                    ((net electricalTypeVar canBeSetTo: ioSignal electricalType)
                     and: [ net datatypeVar canBeSetTo: ioSignal datatype])
                    ifFalse: [ ↑ false ]]]]]
    ↑ true

Variable methodFor: 'test'

canBeSetTo: anObject
"set my value to anObject, propagate, then restore prev values.
returns true if no violation, false otherwise"
...
self lastSetBy: #TENTATIVE.
self prevValue: anObject.
status ← self propagate.
status isNil ifTrue: [ self restore. ↑ false ].
...
self restore.
↑ true

```

Figure 8.2 Test methods

Pruning the search tree is achieved by associating the "ideal" characteristics of subclasses with their (parent) generic cell class. The generic cells then represent the best case estimates of what their descendents can attain. Instead of testing every leaf cell (i.e., non-generic cell) in the class hierarchy, the search process does a pre-order traversal of the search tree, testing each generic cell to determine if its descendents should be tested. If a generic cell fails the tests, then there is no need to test its descendents (Fig. 8.3).

CellWithVarAndConstraints class methodsFor:'module selection'

```
selectRealizationsFor:myInstance priorities:priorityList
|aList|
self isGeneric
ifTrue:[
    aList ← OrderedCollection new.
    self subclasses do:[ :subclass |
        aList addAll:(subclass validRealizationsFor:myInstance
            priorities:priorityList)].
    ↑ aList]
ifFalse:[ ↑ OrderedCollection with:self ]

validRealizationsFor:anInstance priorities:priorityList
"this selects a valid realization for an instance of my ancestor"
self isGeneric
ifTrue:[
    "prune the search tree by testing generic cells as well"
    (self isValidRealizationFor:anInstance priorities:priorityList)
    ifTrue:[ ↑ self selectRealizationsFor:anInstance
        priorities:priorityList ]
    ifFalse:[ ↑ OrderedCollection new ]]
ifFalse:[
    (self isValidRealizationFor:anInstance priorities:priorityList)
    ifTrue:[ ↑ OrderedCollection with:self ]
    ifFalse:[ ↑ OrderedCollection new]]
```

Figure 8.3 Code for Module Selection

For example (Fig 8.4), a generic RippleCarryAdder8, a subclass of Adder8, may have an bounding box area equal to that of its smallest subclass (RCAdd8S) and delay values equal to those of its fastest subclass (RCAdd8F).

In the search for an implementation of a generic Adder8 instance, RippleCarryAdder8 is tested for bounding box and delay properties. If the generic RippleCarryAdder8 is too slow or too big for the intended application, then none of its descendent classes can possibly qualify. So none of the descendents of RippleCarryAdder8 needs to be tested. On the other hand, if the generic class passes the tests, then the search must extend into its subclasses to see if any of them is a valid realization. Early cutoff of the search tree decreases the number of unsuccessful candidates to be tested by testing the best properties of a group of cells as a whole (as represented by the generic cells), thereby increasing the efficiency of the search. However, effectiveness of tree pruning depends heavily on the way cells are organized in the hierarchy and the diligence of the designers to provide the "ideal" estimates for generic cell properties.

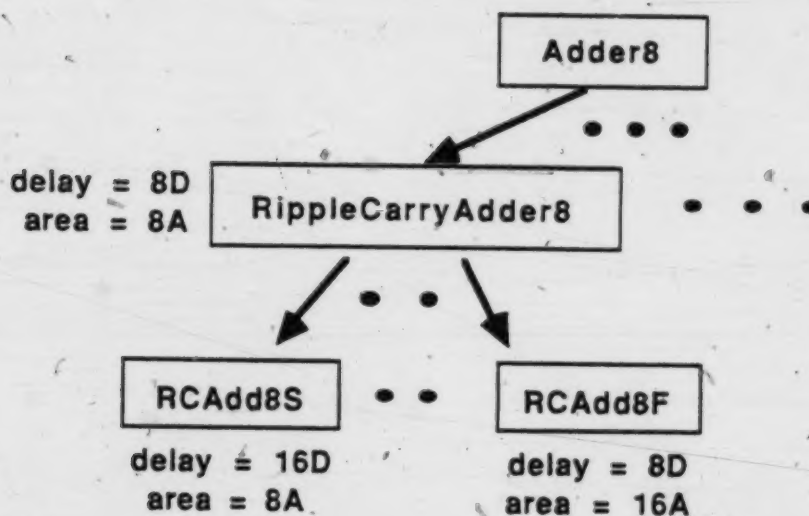


Figure 8.4 Generic class with properties for search pruning

8.3. Summary

In this chapter, generic cells and module selection in STEM are presented. Generic cells allow a least commitment strategy of design by deferring implementation decisions for less critical parts of designs. Combined with automated module selection, generic cells can serve as a vehicle for specifying macro-cell templates that generate custom realizations for different sets of constraints. In STEM, module selection is implemented using the "generate and test" search algorithm, with selective testing and tree pruning to improve the efficiency of the search.

The selection algorithm uses constraint propagation for module validation. This way the validity of a realization depends on all of the constraints in the context in which the generic cell instance is used. Unfortunately, while constraint propagation validates that the characteristics of a cell satisfy the design constraints, it cannot measure how well these constraints are satisfied. A more intelligent module selection algorithm is necessary to differentiate relative merits among different valid realizations of a generic cell. However, this is beyond the scope of this thesis.

Chapter 9. Conclusion

This chapter summarizes this thesis and presents discussions on the features that distinguish STEM's constraint propagation from previous systems, on STEM as a research vehicle for this work, and the limitations of STEM's constraint propagation. Suggestions for future work conclude this thesis.

9.1. Summary

Least-commitment design strategy offers an alternative approach to the traditional top-down and bottom-up hierarchical design styles. By delaying design decisions as much as possible, and refining design specifications whenever characteristics of related designs become available, least-commitment requires fewer design iterations and less redesign efforts in each iteration. However, in order to support the least-commitment design strategy, a design environment must manage interactions among characteristics and specifications of different designs. Toward this goal, this thesis has presented a framework of constraint propagation which manages design interactions in an integrated design environment and supports the least-commitment strategy in IC design.

This framework of constraint propagation consists of a suite of constraint and variable objects organized in an object-oriented type hierarchy. Constraint propagation is triggered by value assignments of variable objects, and is scheduled on fixed priority agendas which help to reduce the amount of redundant computations. Propagation terminates whenever a constraint violation is detected during propagation, or when there are no more active constraints to propagate. Dependency records are kept with propagated values so that dependency analysis can trace all antecedents of a propagated value and all consequences of a constraint or a variable value. New types of constraints and

variables can be easily added to the framework as subclasses of the existing objects, with possibly new methods that specify different propagation behaviors and satisfaction criteria.

This framework of constraint propagation has been integrated into STEM. The dual variables in class and instances of design objects in STEM facilitate the use of implicit constraint variables to link constraint networks in different cells. The resulting system is a framework of hierarchical constraint networks that are able to propagate values from one constraint network to another constraint network, and from one level in the design hierarchy to another level in the design hierarchy. Design characteristics are thereby propagated up the design hierarchy and checked against specifications in other designs. This provides a mechanism for managing interactions among designs and for supporting the least-commitment strategy of IC design.

Constraint propagation has been used in three types of sample applications in STEM. These have been described in this thesis to illustrate different ways constraint propagation can help designers and design tools in an integrated IC design environment such as STEM.

9.2. Discussions

In STEM, a constraint embodies both procedural and declarative knowledge which guides the propagation of variable values through the constraint. This contrasts with purely declarative constraints and purely procedural constraints in other systems. In purely declarative constraints, the semantics of constraints are specified by rules. This limits the semantics of constraints to relationships that can be described by rules. In purely procedural constraints, the context and scope of constraints are embedded in code. This makes

management of constraints very difficult when the number of constraints grow.

Using the object-oriented paradigm, STEM combines the best of both approaches. The semantics of a constraint in STEM are procedurally defined with methods in the constraint object, while the context and scope of the constraint is declared in the connectivities of the constraint to other objects. Propagation of constraints proceeds by applying the procedural knowledge in various constraints in arbitrary orders dictated by the declarative knowledge in these constraints. This mixture of declarative and procedural knowledge representations in STEM's constraints combine the power of procedural computation with the flexibility of declarative dependencies. As a result arbitrary types of constraints can be easily defined, and complex networks of dependencies can be easily added and deleted in STEM.

Constraints allow a user to specify the desired functionality without having to know how it is actually achieved. This is important in an integrated environment in which there is a large number of design tools and a need for effective presentation of these tools to their users, be it designers or other tools in the environment. When used by a designer, constraints provide an explicit representation for intentional design conditions. They allow the designer to specify dependencies and help maintain design conditions. Constraint propagation provides early feedback of design errors so that less efforts are spent on redesigns.

When used by tool developers, constraints hide implementation details of a tool from other tools. The knowledge encoded in the constraints take care of when and how each tool should be invoked so that independently developed tools can make use of one another without really knowing how. This facilitates reuse of tools in the environment, and reduces replication of functionality in

different tools.

9.2.1. Distinctions from Previous Work

The research reported in this thesis builds upon previous work on constraint propagation, especially ThingLab and CONSTRAINTS. However, while the majority of past research has focused on pure constraint propagation systems, this thesis has focused on using constraint propagation as a support mechanism in an integrated IC design environment. As a result, there are major differences in the implementation between STEM's framework of constraint propagation and that of other constraint systems. In particular, two implementation details distinguish STEM's framework of constraint propagation. The first is the way STEM uses dual variables and implicit constraint variables to implement hierarchical constraint propagation. This leads to STEM's hierarchical constraint networks that parallel the design hierarchies (Fig. 9.1). These hierarchical networks take advantage of the inherent structures in STEM's design hierarchy to reduce redundant propagation of constraints.

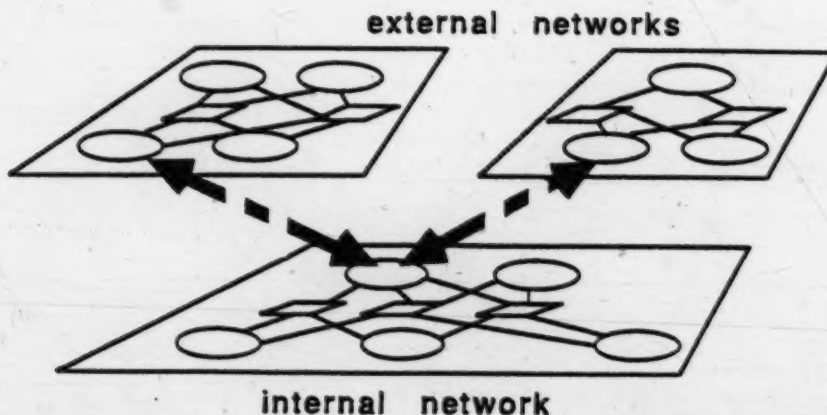


Figure 9.1 Two level hierarchical constraint networks

The second implementation detail that distinguishes STEM's constraint propagation framework is that propagation knowledge is encoded in methods of constraint and variable objects and inherited by their subclasses. The framework defines a protocol of messages that each constraint and variable respond to, and abstracts common propagation knowledge to methods in a few primitive classes of constraints and variables. This object-oriented implementation has two advantages: new types of constraints and variables can be easily added to the system as subclasses of the existing types, and arbitrarily complex constraint and variable behaviors can be specified by overwriting the inherited methods in these subclasses.

9.2.2. Comments on STEM

STEM has served well as a vehicle for this research. Its underlying object-oriented programming environment, Smalltalk-80, is especially well suited to rapid prototyping. Despite the initial learning curve for the message-passing programming paradigm, object-oriented programming provided high flexibility and efficiency in coding and debugging. Heavy reuse of code was encouraged by protocol inheritance and greatly facilitated by Smalltalk's browsers and multi-window user interface. The interpretive debugger in Smalltalk also proved to be an invaluable programming and debugging aid.

The organization of STEM's data structures was instrumental in the implementation of hierarchical constraint propagation. While the general framework of constraint propagation presented in this thesis can be easily implemented in other object-oriented design environments, it is more difficult to implement STEM's hierarchical constraint propagation in other systems since it requires dual declaration of instance variables in the class and instance definitions of

each design object.

9.2.3. Limitations of STEM's Constraint Propagation

There are inherent limitations to the capabilities of STEM's constraint propagation. First, activated constraints are not scheduled by dependencies due to performance concerns. This creates problems for reconvergent fanouts because of the one-value-change rule for variable objects. If a constraint is propagated after its dependent constraints are propagated, some variables would likely have incorrect values. A quick fix is to relax the one-value-change rule to allow N value changes in each propagation cycle, N being heuristically determined from the network. The correct solution is to develop an efficient algorithm that dynamically schedules activated constraints by dependencies.

Second, since each constraint propagates based entirely on localized data, optimization type problems which require global considerations cannot be solved by constraint propagation. Constraint satisfaction, which examines constraints on a global scale, is required for these problems. Even for problems that can be solved by constraint propagation, there is the issue of applicability due to performance concerns.

Currently, the size of manageable constraint networks in STEM is limited by both speed and storage inefficiencies of Smalltalk. This is due largely to the way Smalltalk is designed. Smalltalk execution is slowed by the overhead of message passing, dynamic type checking, and run-time interpretation of "byte codes" that are the primitive instructions for the Smalltalk virtual machine. The storage inefficiency is due to the extremely fine granularity of Smalltalk's objects. Unlike other object-oriented programming environments, Smalltalk implements almost everything in the environment as an object and incurs

significant storage overhead.

Nevertheless, constraint propagation imposes performance penalties that limit its applicability in an interactive environment. The time and storage complexity of STEM's constraint propagation is of an order proportional to the summation of the number of constraints over all variables in the network.

$$\text{complexity} \propto \sum_v \text{numberOf ConstraintsOf } v$$

Consequently, constraint networks should be small and sparse. For large and dense networks like layout constraints, specialized data structures, such as corner stitching, and problem specific algorithms, such as graph based compaction algorithms, are required to achieve the necessary performance. Constraint propagation is best suited for deriving and checking higher level properties of cells like delay, signal types and bounding box. These constraint networks tend to be sparse and highly localized such that propagation of constraints can be efficiently applied.

9.3. Suggestions for Future Work

Future research in constraint propagation in STEM can progress in several directions. First, a good user interface for browsing, specifying and editing of constraints is needed. The user interface may include a graphical display of constraint networks, a text window that explains the semantics of selected constraints, and a form window for editing the connectivities of selected constraints. This user interface can then be linked to the constraint debugger, and be brought up whenever constraints are violated to provide diagnostic explanations and recommended actions to the user.



Second, a higher degree of control can be implemented for enabling and disabling constraint propagation. For example, disabling propagation and/or checking of individual constraints, constraints in particular networks, specified types of constraints, and constraints connected to specific sets of variables. A sophisticated user interface may be needed to guide the selection of constraints for enabling and disabling.

Third, constraint networks can be compiled to improve the efficiency of constraint propagation. Compilation of constraint networks can take several forms, ranging from simple topological sorts of the constraint networks to complete proceduralization of the constraints. The difficult issues are deciding which constraints should be compiled and what kind of compilation is most suitable. A correct mix of declarative and procedural implementation of constraints must balance run-time efficiency with manageability of the networks.

Fourth, the natural extension to propagation of constraints is probably a framework of constraint satisfaction. Unlike constraint propagation which only examines localized data when making each decision, constraint satisfaction attempts to solve a constraint network by global considerations. While algorithmic constraint satisfaction techniques are well known for well defined numerical constraints, they do not seem readily extensible to handle arbitrary constraints such as those found in STEM. An object-oriented approach to constraint satisfaction may just be the way to solve for networks of arbitrary constraints.

Finally, the example applications presented in this thesis can be greatly improved. Since this research focused primarily on STEM's constraint propagation framework and only developed these applications to demonstrate this framework, these applications did not receive the emphasis that they deserved.

In fact, each of the applications can be a good research project. In particular, module selection is becoming more and more important as the number of alternative designs in cell libraries grows and as the design space increases. More intelligent and efficient module selection techniques are required to help explore larger design space more quickly in order to improve the quality of designs.

References

- [Born77] A. Borning, "ThingLab -- An Object-Oriented System for Building Simulations Using Constraints", Proceedings, Fifth International Joint Conference on Artificial Intelligence(IJCAI-5), MIT, Cambridge, Aug 1977, pp497-498.
- [Brow83] H. Brown, C. Tong & G. Foyster, "Palladio: an Exploratory Environment for Circuit Design", IEEE Computer, Dec. 1983, pp41-56.
- [Garl86] D. Garlan, "Views for Tools in Integrated Environments", *Advanced Programming Environments*, Lecture Notes in Computer Science 244, Springer-Verlag, 1986.
- [Girc87] E. F. Girczyc & T. A. Ly, "STEM: An IC Design Environment Based on the Smalltalk Model-View-Controller Construct", Proceedings, 24th Design Automation Conf., 1987.
- [Gold83] A. Goldberg & D. Robson, *Smalltalk-80 The Language and Its Implementation*, Addison-Wesley, Don Mills, Ontario, 1983.
- [Katz83] R. H. Katz, "Managing the Chip Design Database", IEEE Computer, Vol 16, No. 12, Dec. 1983, pp26-35.
- [Kell84] V. E. Kelly, "The CRITTER System", Proceedings, 21st Design Automation Conference, 1984.
- [Law85] H-F.S. Law & J.D. Mosby, "An Intelligent Composition Tool for Regular and Semi-Regular VLSI Structures", IEEE ICCAD 1985 Proceedings, pp169-171.
- [Ly87] T. A. Ly, R. Miller & E. F. Girczyc, "Interfacing Application Programs to an Object-Oriented Database Using Views and Controllers", Proceedings, CCVLSI-87, 1987, pp87-92.
- [Mayo86] R. N. Mayo, "Mocha Chip: A System for the Graphical Design of VLSI Module Generators", IEEE ICCAD 1986 Proceedings, pp74-77.
- [Mitt85] T. M. Mitchell, L. I. Steinberg & J. S. Shulman, "A Knowledge-Based Approach to Design", IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-7, No. 5, September 1985, pp502-510.
- [Oust83] J. K. Ousterhout, "Crystal: A Timing Analyzer for NMOS VLSI Circuits", *Proceedings of the Third Caltech VLSI Conference*, 1983, pp57-59.
- [Oust85] J. K. Ousterhout, et al, "The Magic VLSI Layout System", IEEE Design & Test, February 1985, pp19-30.
- [Pulv87] M. Pulver & M. I. Elmasry, "Using Igloo: A Constraint Based Layout Language for VLSI", Proceedings, CCVLSI-87, 1987, pp81-86.
- [Rubi87] S. Rubin, *Computer Aids for VLSI Design*, Addison-Wesley, Don Mills, Ontario, 1987.
- [Sale84] R. Saleh, "Iterated Timing Analysis and SPLICE1", Electronics Research Laboratory, University of California, Berkely, Memo No. UCB/ERL M84/2, January, 1984.
- [Stall77] R. M. Stallman & G. J. Sussman, "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis", *Artificial Intelligence* 9, 1977, pp135-196.

- [Stef81] M. Stefik, "Planning with Constraints (MOLGEN: Part 1)", Artificial Intelligence, Artificial Intelligence 16, 1981, pp111-140.
- [Suss80] G. J. Sussman & G. L. Steele Jr., "CONSTRAINTS- A Language for Expressing Almost-Hierarchical Descriptions", Artificial Intelligence 14, 1980, pp1-39.
- [VICP74] A. Vladimirescu, E. Cohen, D. O. Pederson, "SPICE2 User's Guide", Electronics Research Laboratory, University of California, Berkley, April 1974.
- [Wied86] G. Wiederhold, "Views, Objects, and Databases", IEEE Computer, Dec. 1986, pp37-44.
- [Wolf86] W. Wolf, "An Object-Oriented, Procedural Database for VLSI Chip Planning", Proceedings, 23rd Design Automation Conf., 1986.